UNIVERSITÀ DI PISA

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

PROGETTAZIONE E SVILUPPO DI UN ALGORITMO DI CLUSTERING BASATO SULLA DENSITÀ IN UNO SPAZIO RELAZIONALE AD ALTA DIMENSIONALITÀ E CONFRONTO CON ALGORITMI SIMILI PRESENTI IN LETTERATURA

RELATORI

Ch.mi Proff.

Beatrice Lazzerini

Francesco Marcelloni

Mario Giovanni C. A. Cimino

CANDIDATO

Davide Bolognesi

Premessa

I sistemi informatici aziendali odierni dispongono di capacità di archiviazione e varietà di informazione sempre crescenti. Grazie anche alla connettività fornita dalla rete Internet, le risorse informative da gestire aumentano a dismisura. Nasce quindi l'obiettivo strategico di poter acquisire anche la conoscenza (informazione ad alto livello d'astrazione) desumibile da tale informazione. A tale scopo, uno dei settori più promettenti della ricerca è il Data-Mining ovvero l'insieme di principi e tecniche utili ad 'estrarre' conoscenza dai dati. In particolare, si consideri la cluster analysis il cui obiettivo è quello di individuare in un insieme di oggetti, rappresentati come vettori in uno spazio di caratteristiche reali, una organizzazione in categorie. Un algoritmo di clustering (categorizzazione) è di tipo 'oggetto' se opera su vettori le cui caratteristiche rappresentano misurazioni dirette sugli oggetti, è di tipo 'relazionale' se tali caratteristiche rappresentano dissimilarità dell'oggetto rispetto a tutti gli altri elementi dell'insieme di dati, quindi in uno spazio ad alta dimensionalità. Tipicamente, gli algoritmi di tipo oggetto adoperano una metrica nello spazio di rappresentazione, e per questo non possono essere efficacemente applicati per alte dimensionalità; mentre gli algoritmi relazionali non hanno una convergenza garantita a meno che la dissimilarità non abbia proprietà particolari.

Argomento della presente tesi è la progettazione di un algoritmo relazionale per alta dimensionalità che converga

anche per dissimilarità prive di particolari proprietà metriche, e che sia basato su argomenti di densità. Dopo la descrizione delle soluzioni esistenti e la progettazione dell'algoritmo ROPTICS, la trattazione prosegue con la fase di sviluppo in linguaggio C++ ed il confronto, su insiemi di dati di riferimento, con soluzioni simili presenti in letteratura.

Indice generale

Premessa ii

Capitolo 1

Introduzione 1
1.1 Fondamenti teorici sul clustering o categorizzazione 2
1.2 Tecniche di clustering 6
1.2.1 Prestazioni di un algoritmo di clustering 11
1.3 Applicazioni del clustering 12

Capitolo 2

Analisi del problema 13
2.1 Il problema dell'alta dimensionalità 13
2.2 Algoritmi esistenti: DBSCAN 15
2.3 Algoritmi esistenti: OPTICS 17
2.3.1 Individuazione dei cluster 24
2.4 Fondamenti teorici del nuovo algoritmo 29

Capitolo 3

Progettazione di ROPTICS 32
3.1 Operazioni svolte 32
3.2 Ulteriori aspetti rilevanti 35
3.3 Complessità asintotica 38

Capitolo 4

Sviluppo dell'applicazione 40
4.1 Identificazione di oggetti e servizi 40

Capitolo 5

Risultati sperimentali 42
5.1 Test su dati sintetici 42
5.2 Test su dati reali 45
5.3 Conclusioni 49

Capitolo 6

Descrizione di algoritmi analoghi 50
6.1 FCM - Fuzzy C-Means 50
6.2 RFCM – Relational FCM 52
6.3 FCMdd – Fuzzy C-medoids 52
6.4 R-NE-FRC - Robust Non-Euclidean Fuzzy Relational data Clustering 53
6.5 NeRFCM – Non Euclidean Relational FCM 55
6.6 ARCA – Any Relational Clustering Algorithm 55
6.7 FNM – Fuzzy Non Metric Model 56
6.8 AP – Assigment-Prototype 57

Capitolo 7

Confronto tra i vari algoritmi 58
7.1 Confronto con FCM 58
7.2 FCM ed estrazione delle somiglianze 62
7.3 Confronto con NE-FRC 97

7.4 Confronto con altri algoritmi..... 100
7.5 Considerazioni riassuntive..... 108
Capitolo 8
Conclusioni e sviluppi futuri..... 110
Appendice A
Ulteriori dettagli implementativi..... 112
A.1 Analisi dei requisiti..... 112
A.2 Sintassi e regole di funzionamento..... 112
A.3 Descrizione delle classi..... 114
Appendice B
Codice sorgente..... 116
Bibliografia..... 158

Capitolo 1

Introduzione

La quantità e la varietà di informazione presente nei sistemi informativi aziendali cresce incessantemente. I file system (sistemi di gestione dei file) ed i DBMS (sistemi di gestione dei dati), offrono la possibilità di archiviare ed organizzare le informazioni in modo strutturato ed efficiente. Una tale mole di dati costituisce un potenziale patrimonio in termini di conoscenza (cioè informazioni ad alto livello di astrazione) ma, date le dimensioni, non è facilmente estraibile tramite le classiche tecniche di analisi statistica. A tale proposito, vi sono discipline di ricerca che studiano lo sviluppo di strumenti per l'interpretazione dei dati nei diversi contesti applicativi. Con riferimento al Data Mining, tra i metodi di estrazione di conoscenza, vi sono le tecniche del clustering oppure le regole di associazione. Per quanto riguarda la prima categoria, le applicazioni sono le più disparate. Un esempio importante è rappresentato dal Text Mining ossia la metodologia volta alla ricerca, analisi e classificazione tematica dei documenti. Come scenario applicativo, si pensi alla possibilità di analizzare in maniera automatica i file di log[1] dei server web in modo da scoprire le tipologie di visitatori del sito. Le regole di associazione sono molto utili, ad esempio, in ambito commerciale. Nell'ambito applicativo si utilizza la cosiddetta 'market basket analysis', in cui a partire dalla composizione della spesa di ogni cliente si ottengono informazioni sulle associazioni tra vari prodotti, consentendo così di programmare promozioni o campagne pubblicitarie redditizie. Nel seguito si farà sempre riferimento al clustering.

1 I file di log sono comunemente usati per tenere traccia delle pagine accedute dagli utenti durante una sessione di navigazione.

1.1 Fondamenti teorici sul clustering o categorizzazione

Occorre, a questo punto, introdurre alcune definizioni comunemente adottate nell'ambito di cui si sta trattando. Si definisce *pattern* (oppure *modello*, *vettore di feature*, *osservazione*, *dato*) una singola entità informativa; generalmente se i pattern sono rappresentati da punti in $\mathbb{R}^D$ la notazione che si usa è la seguente:

$$\boldsymbol{x} = (x_1, \ldots, x_D)$$

Ogni coordinata x_i è detta *feature* ovvero *attributo*, D è detta *dimensionalità* del pattern; si vedrà in seguito che i singoli pattern non sono sempre rappresentati dai loro attributi. Un attributo può assumere uno di questi tre tipi di valore:

- **binario:** può assumere un valore booleano vero o falso
- **discreto:** può assumere un numero finito di valori
- **continuo:** può assumere un numero infinito di valori

Inoltre un attributo può essere:

- **Qualitativo:** questa tipologia si può dividere in due sottoinsiemi:
    - **Nominale:** i vari valori sono espressi con etichette, senza un ordinamento, ad esempio “rosso”, “verde”, “blu”.
    - **Ordinale:** i valori possiedono un ordinamento ad esempio “sufficiente”, ”buono”, “ottimo”.
- **Quantitativo:** comprende i seguenti sottoinsiemi:
    - **Intervallo:** è significativa la differenza tra i valori, ad esempio valori di temperatura nella scala Celsius.
    - **Rapporti:** sono significativi i rapporti tra i vari valori poiché esiste uno zero assoluto ad esempio la temperatura nella scala Kelvin o la misura di corrente elettrica.

È evidente che i valori non numerici dovranno essere opportunamente trasformati per adattarsi alle procedure di calcolo.

Si definisce *classe* un aggregato di enti caratterizzati da una data estensione di una proprietà, determinata da una legge 'naturale' che stabilisce la semantica di tali enti in quanto elementi della classe; ciò che si intende fare è trovare un modo automatico per raggruppare nello stesso insieme i pattern generati in base alla stessa legge.

Con *clustering (o categorizzazione)* si indica la partizione non supervisionata dei pattern in gruppi detti *cluster*. L'organizzazione delle categorie avviene in base ad una misura di somiglianza tra pattern: pattern più simili appartengono alla medesima categoria mentre pattern dissimili appartengono a categorie diverse. Nel *clustering* di un insieme di oggetti viene prodotto un raggruppamento "anonimo", mentre nella *classificazione* di un oggetto i gruppi sono (parzialmente) già definiti ed i relativi pattern (training pattern) possiedono una etichetta che denota la semantica della classe. Classificare un pattern significa assegnarlo ad una delle possibili classi. In tal senso, si dice che il clustering è una classificazione non supervisionata.

Al fine di chiarire meglio i concetti e le definizioni precedenti viene fornito un esempio; si consideri un gruppo di punti in $\mathbb{R}^2$, detto insieme di dati (o data set), rappresentati in figura 1.

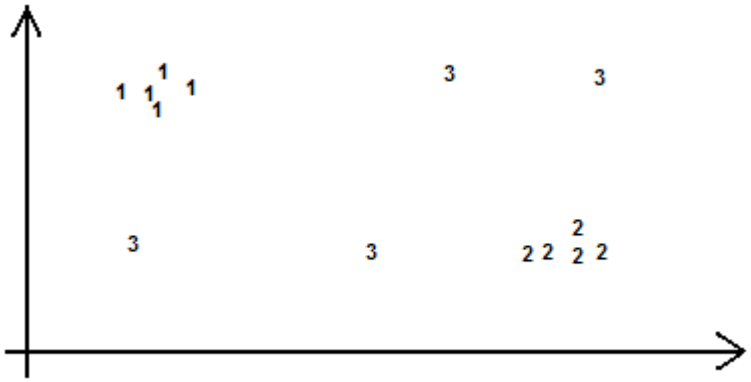

figura 1: Rappresentazione di un insieme di dati nel piano

È facilmente intuibile che i punti con etichetta 1 appartengono ad un cluster, quelli etichettati con 2, fanno parte di un altro cluster, mentre presumibilmente i punti 3 non sono significativi; questi ultimi sono chiamati *rumore* o *outlier*. Un algoritmo di clustering, quindi, consente di formare i cluster; si noti che in problemi reali spesso si ha a che fare con migliaia di dati in decine di dimensioni, in tal senso l'ausilio di uno strumento automatico si rivela indispensabile.

Effettuare il clustering su una certa quantità di dati reali prevede, generalmente, i seguenti compiti:

- Rappresentare i pattern: riguarda la scelta del numero dei pattern, delle dimensioni e del tipo delle feature. Scegliere il giusto numero di feature, o effettuare trasformazioni su di esse, può portare a significativi miglioramenti sul risultato del procedimento. Tale fase può comprendere, eventualmente anche i seguenti passi:
    - Feature selection: consiste nel selezionare un sottoinsieme significativo delle feature di cui si dispone; si noti che, in generale, non è banale riuscire a capire a priori quali siano gli attributi migliori ai fini del risultato.
    - Feature extraction: opera trasformazioni sulle feature in ingresso al fine di trovarne di nuove e più significative.
- Definizione di una misura di prossimità: per poter raggruppare i pattern in insiemi omogenei occorre quantificare la diversità tra i vari elementi (spesso si tratta di una distanza tra punti nello spazio come la distanza euclidea).
- Raggruppamento (grouping o clustering): è la procedura che permette di effettuare la categorizzazione. Si vedrà in seguito che essa può avere varie implementazioni.
- Astrazione dei dati: consiste nell'estrarre una rappresentazione semplice e compatta dei dati. Tale rappresentazione deve essere sia comprensibile all'uomo, sia facile da elaborare. Un tipico esempio di astrazione è il fatto che un cluster può essere rappresentato per mezzo di un punto detto prototipo.
- Verifica dei risultati: per stimare la bontà dei risultati prodotti occorre avere un criterio di ottimalità; una struttura di cluster è detta *valida* se non è stata prodotta dal caso o da un artefatto interno all'algoritmo. Ci sono tre modi di accertare la validità:

    - Verifica esterna: Si controlla se la struttura dei dati ricostruita dall'algoritmo rispetta la struttura nota a priori.
    - Verifica interna: Si valuta che la struttura ottenuta dal clustering sia appropriata alla tipologia di dati.
    - Verifica relativa: Date due strutture di dati ricostruite dall'algoritmo, ne viene calcolata la qualità relativa.

Come si è visto, nella categorizzazione dei dati gioca un ruolo fondamentale l'introduzione di una misura della somiglianza dei vari pattern; a tale proposito viene comunemente utilizzato il concetto di *dissimilarità*; il valore di questa quantità viene spesso fatto coincidere con una qualche metrica di distanza come la distanza euclidea:

$$d_2(x,y)=(\sum_{k=1}^{M}(x_k-y_k)^2)^{\frac{1}{2}}=\|x-y\|_2$$

Questa formula rappresenta un caso particolare della metrica di Minkowski riportata di seguito:

$$d_p(x,y)=(\sum_{k=1}^{M}(x_k-y_k)^p)^{\frac{1}{p}}=\|x-y\|_p$$

In cui *M* è il numero di dimensioni dello spazio considerato.

Generalmente tale distanza si usa negli spazi a due o tre dimensioni e funziona tanto meglio quanto i cluster sono separati e compatti. L'inconveniente tipico dell'uso della metrica di Minkowski riguarda la possibilità che una delle feature sia definita in un intervallo molto più ampio delle altre; in questi casi si deve ricorrere alla normalizzazione dei dati. Un altro inconveniente si presenta quando le feature hanno una certa correlazione lineare; per limitare quest'ultimo problema (che rende distorte le misure di distanza) si ricorre ad una definizione alternativa di distanza, detta di Mahalanobis così definita:

$$d_m(x,y)=(x-y)\sum^{-1}(x-y)^T$$

in cui i pattern *x* ed *y* sono vettori colonna e Σ rappresenta la matrice di covarianza ottenuta tramite campionamento dei pattern o dal processo di generazione dell'insieme di dati; per ogni coppia di pattern vengono assegnati diversi pesi alle feature in base ai valori di varianza e di correlazione lineare tra la coppia. Le precedenti misure di distanza sono utili nel caso di cluster ipersferici (l'euclidea) o iperellissoidali (di Mahalanobis).

Come si è detto la rappresentazione dell'insieme di dati tramite nello spazio delle feature non è la sola possibile; nel caso in cui la dimensionalità sia limitata (fino ad una decina di feature) i pattern possono essere rappresentati efficacemente tramite una matrice contenente le distanze reciproche; chiameremo quest'ultima *matrice di dissimilarità*.

1.2 Tecniche di clustering

Le varie tecniche di categorizzazione differiscono per i diversi approcci utilizzati e per la tipologia dei risultati; tali tecniche possono essere concettualmente organizzate ad albero.

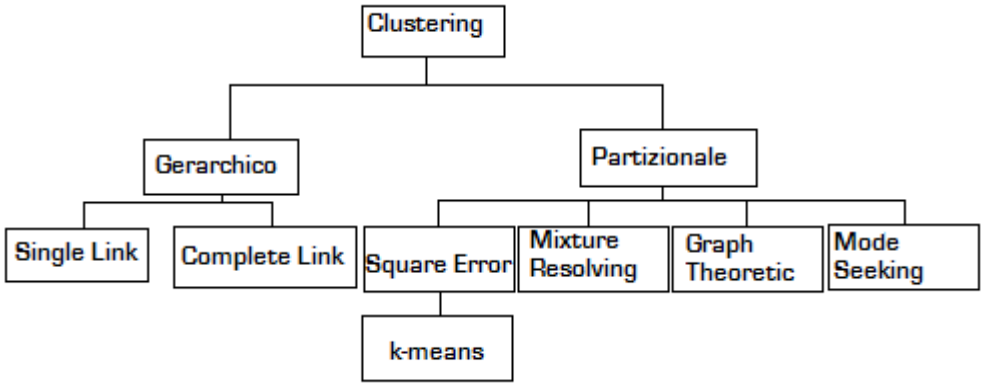

figura 2: Principali tecniche di clustering

Il clustering può essere:

- Agglomerativo-Divisivo: nel primo caso si assegna ogni pattern ad un cluster diverso e si agglomerano via via i cluster più vicini; nel caso divisivo si ha, inizialmente, un unico cluster contenente tutti i pattern e lo si divide per passaggi successivi.
- Monotetico-Politetico: nel primo caso l'algoritmo utilizza una feature per volta; nell'altro tutte le feature entrano in gioco contemporaneamente.
- Hard-Fuzzy: dati C cluster; un algoritmo hard assegna ad ogni punto una etichetta che rappresenta il cluster a cui il punto appartiene mentre un algoritmo fuzzy assegna ad ogni pattern C valori che rappresentano i gradi di appartenenza del pattern ad ogni cluster; la somma di questi valori è 1.
- Rappresentazione Relazionale-Oggetto: gli algoritmi relazionali utilizzano matrici di dissimilarità, e si basano unicamente sul concetto di vicinanza inter-oggetto; mentre l'approccio opposto consiste nel rappresentare un pattern con le sue M feature cioè come un punto in $\mathbb{R}^M$. L'utilizzo di una rappresentazione o l'altra dipende dalle informazioni a disposizione sui dati e dalla disponibilità di metriche valide per lo spazio di rappresentazione.

Tornando alla classificazione mostrata in figura 2, si definisce *clustering gerarchico* un

algoritmo che produce raggruppamenti annidati; si definisce *clustering partizionale* un algoritmo che crea una unica partizione dell'insieme di dati. Un esempio di clustering gerarchico agglomerativo è costituito dal metodo Single-Link: si parte mettendo ogni oggetto in un cluster e si aggregano via via i due cluster più somiglianti fino ad avere un unico cluster. Per valutare la somiglianza tra due cluster si valutano le distanze tra tutte le coppie di punti e se ne prende la minore. Si consideri, ad esempio la figura seguente:

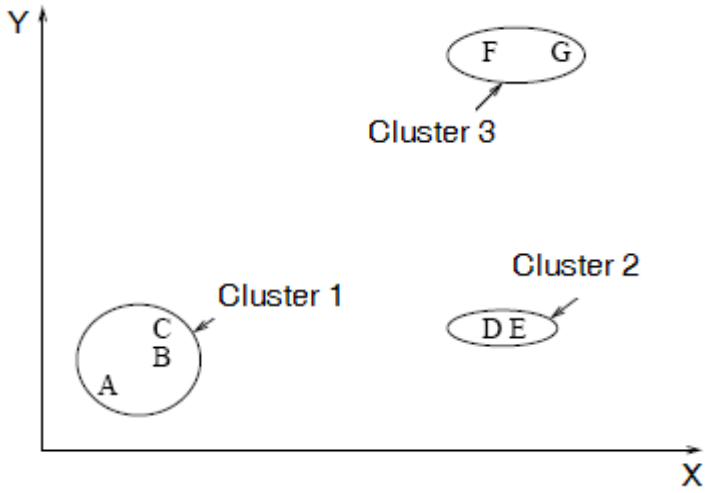

figura 3: Semplice data set

Tramite il Single-Link si ottiene una categorizzazione gerarchica schematizzabile tramite il cosiddetto *dendrogramma* cioè un albero secondo il quale l'insieme di dati viene suddiviso in sottoinsiemi finché questi non posseggono un solo pattern.

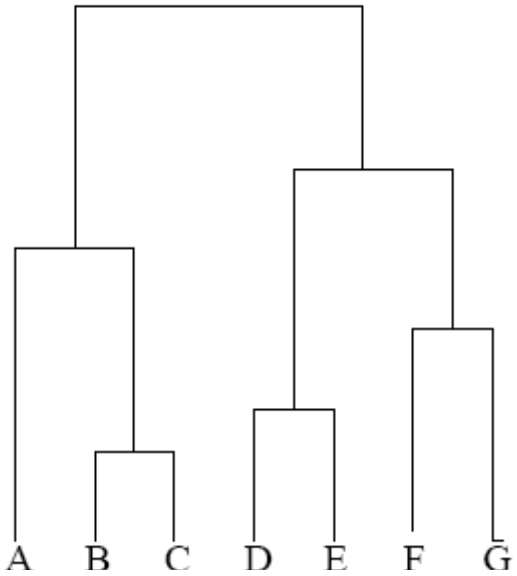

figura 4: Risultati del clustering Single- Link effettuato sull'esempio di figura 3

Il problema principale che rende poco affidabile questa soluzione è il cosiddetto effetto Single-Link, cioè se cluster anche molto densi sono collegati da una sottile striscia di punti, non sono separabili. Come si vede dalla figura seguente non si riesce ad avere una rappresentazione adeguata.

figura 5: Effetto Single-Link

L'alternativa consiste nel ricorrere al cosiddetto Complete-Link, in cui si valuta la

somiglianza tra le varie coppie di cluster in base alla massima tra tutte le distanze tra tutte le possibili coppie di pattern. Come appare chiaro, tali algoritmi sono computazionalmente costosi, ed il dendrogramma può risultare di difficile interpretazione su insiemi di punti molto grandi.

L'altra grande famiglia di metodi di clustering è data dagli algoritmi partizionali basati sull'ottimizzazione di una opportuna funzione di costo, in cui ogni cluster è rappresentato da un *prototipo* detto anche *centroide*; i vari pattern sono assegnati al cluster che ha il centroide più vicino. Minimizzando in modo iterativo la funzione obiettivo, l'algoritmo identifica i cluster in modo tale che i prototipi si posizionino laddove vi sono raggruppamenti più compatti e separati. Infatti, si minimizza una certa misura di inadeguatezza tra i prototipi e gli oggetti. Questi algoritmi si prestano bene alla ricerca di cluster convessi nel caso in cui il numero degli stessi sia facilmente determinabile, inoltre sono preferibili agli algoritmi gerarchici nel caso in cui il grande numero di pattern renda difficoltosa la costruzione del diagramma.

Una ulteriore suddivisione distingue la famiglia di algoritmi partizionali in tre categorie: *k-means, k-modes*, *k-medoid.* Nel *k-means* le coordinate del centroide sono equivalenti a quelle del baricentro dei punti appartenenti al cluster; il *k-modes* estende il precedente ai domini categorici[2]; per quanto riguarda il *k-medoid*, i prototipi, detti appunto *medoid*, sono rappresentati da un pattern vicino al centro del cluster. Questi algoritmi nella maggior parte dei casi sono utilizzati nella loro versione fuzzy. L'approccio fuzzy consiste nell'assegnare diversi gradi di appartenenza di un pattern rispetto a vari cluster tale che la somma dei gradi di appartenenza dello stesso pattern ai vari cluster vale uno. La matrice delle appartenenze di tutti i pattern viene detta *partizione fuzzy*; volendo passare ad una rappresentazione non fuzzy, detta comunemente *crisp*, occorre procedere alla *defuzzificazione* della partizione ottenuta. Normalmente si assegna il pattern al cluster rispetto al quale il grado di appartenenza è maggiore.

Come si è visto al termine di una procedura di clustering si deve accertare la qualità dei risultati ottenuti; occorre ricordare, infatti, che nei casi pratici non si conosce la corretta classificazione e spesso neanche il numero esatto di cluster.

Al fine di valutare la qualità della categorizzazione ottenuta, sono stati introdotti degli indici di qualità calcolabili a partire dai risultati. Uno degli indici più usati per tecniche basate su matrice di appartenenza fuzzy è il *coefficiente di partizione* così definito:

2 Per dominio categorico si intende un insieme di valori sul quale non ha senso effettuare operazioni matematiche o esprimere predicati a parte l'uguaglianza.

$$V_{PC}(U,C)=\frac{1}{N}\sum_{k=1}^{N}\sum_{i=1}^{C}u_{ki}^2$$

Il coefficiente di partizione è funzione della matrice dei gradi di appartenenza U e del numero di cluster C. Intuitivamente l'indice indica quanto è netto e preciso il clustering effettuato, ad esempio se i vari punti appartengono al rispettivo cluster con grado di appartenenza pari circa ad uno (e quindi circa zero nei confronti degli altri cluster) il coefficiente di partizione sarà vicino all'unità (c'è una partizione quasi crisp); se, invece, i vari punti presentano, rispetto ai vari cluster gradi di appartenenza simili (al limite $1/C$) il coefficiente sarà basso di conseguenza (c'è maggiore incertezza). Altri due indici molto usati sono l'*entropia* della partizione e l'indice di *Xie-Beni*. Il primo è definito come segue:

$$V_{PE}(U,C)=-\frac{1}{N}\sum_{k=1}^{N}\sum_{i=1}^{C}u_{ki}\log_a u_{ki}$$

Il significato della precedente può essere sintetizzato in questi termini: se i punti appartengono ad un cluster con un grado di appartenenza vicino ad uno ed agli altri con grado vicino allo zero, l'indice di entropia sarà prossimo allo zero; altrimenti, se i gradi di appartenenza sono prossimi ad *1/C* (massima incertezza), l'indice tenderà a $\log_a C$. L'indice di Xie-Beni, infine, è il rapporto tra l'indice di compattezza:

$$comp=\frac{1}{N}\sum_{m=1}^{C}\sum_{i\in G_m}\|\bar{x}_i-\bar{c}_m\|^2$$

(dove G_m è l'insieme degli indici dei punti del cluster m) e l'indice di separazione:

$$sep=\min_{i\neq j}\|\bar{c}_i-\bar{c}_j\|$$

dove c_i rappresenta il vettore delle feature del baricentro del cluster i.

Qualora il numero di cluster sia ignoto, si effettuano diversi tentativi e, in base al valore di uno degli indici presentati, si assume che i dati siano caratterizzati dal numero di cluster per cui l'algoritmo ha prodotto la partizione di migliore qualità (in termini di coefficiente di partizione, ad esempio).

L'ultima tipologia di algoritmi che viene presa in esame riguarda il clustering basato sulla densità. Secondo tale approccio, i cluster sono identificati come regioni dello spazio ad alta densità e separati da regioni a densità minore (i pattern ivi contenuti sono trattati come rumore). La caratteristica più interessante ed utile consiste nel fatto che queste regioni possono avere forma qualunque (a differenza degli algoritmi partizionali che identificano cluster di una forma predefinita). Il modo più tradizionale di effettuare

questa categorizzazione consiste nel suddividere lo spazio in cui sono definiti i dati in celle non sovrapposte (grid-based clustering); in questo modo, costruendo un istogramma relativo alla densità, le celle potenzialmente appartenenti ad un cluster saranno rappresentate dai picchi del diagramma, mentre le celle contenenti solo rumore saranno quelle in cui il grafico non presenta picchi. Gli algoritmi che saranno descritti in dettaglio nel seguito sono basati sulla densità ma non sono grid-based, bensì valutano per ogni punto la densità dei punti ad esso vicini.

1.2.1 Prestazioni di un algoritmo di clustering

Un questione molto importante nell'ambito del clustering riguarda la *scalabilità* cioè la possibilità di elaborare una maggiore numerosità o dimensionalità di dati mantenendo l'efficacia e l'efficienza. In termini teorici per complessità di un algoritmo si indica la funzione che associa alla dimensione del problema il costo della sua risoluzione in base ad una misura quale, ad esempio, il tempo di esecuzione o lo spazio di memoria occupato. Con classe di complessità si indica la più piccola funzione tra quelle che approssimano la misura ottenuta secondo la metrica scelta. Per comprendere ciò si consideri la complessità del *k*-means: dati C cluster, N pattern e T numero massimo di iterazioni; si dimostra che la classe di complessità temporale è $O(NCT)$.[3] È chiaro che tale valore rende inutilizzabile l'algoritmo per dati molto numerosi; inoltre si noti che spesso C non è noto a priori per cui possono rendersi necessarie varie prove per capire quale sia il suo valore ottimale.

Un'altra problematica comune riguarda la *complessità spaziale* cioè l'ingombro dei dati. La memoria principale, in genere, non sarà in grado di mantenere tutti i dati; occorre, quindi, fare in modo da minimizzare lo spostamento delle pagine dalla memoria di massa. Per fare ciò si considerano queste tre possibili strategie:

1. Si memorizzano i dati nella memoria di massa e si effettua il clustering su un sottoinsieme dei dati; alla fine si aggiungono i pattern non elaborati al cluster più vicino (approccio *divide et impera*).
2. Si memorizzano i dati in memoria di massa e si trasferiscono uno alla volta in

3 Si considerino due funzioni $f,g: N \rightarrow N$. Si dice che $g(n)$ è di ordine $O(f(n))$ se esistono un intero n_0 ed una costante $c>0$ tali che per ogni $n \geq n_0 \quad g(n) \leq cf(n)$

Nel caso in cui l'espressione si riferisca ad una classe di complessità temporale indica la più piccola funzione che approssima la durata temporale del programma.

memoria principale, le informazioni sulla struttura dei cluster vengono anch'esse mantenute in memoria principale.

3. Si effettua l'elaborazione su una macchina parallela. In questo caso i benefici che si ottengono sono in funzione della struttura dell'algoritmo[4].

1.3 Applicazioni del clustering

La cluster analysis è nata negli anni '70 e, tipicamente, era usata nell'ambito della statistica. Oggi si assiste, grazie anche alle elevate capacità di calcolo degli elaboratori attuali, ad una rapida diffusione dell'utilizzo di queste tecniche. Tra gli altri, si ricordano i seguenti campi applicativi:

- *Image segmentation:* data una immagine reale la si vuole confrontare con un database di immagini; per fare ciò si applica un algoritmo di clustering all'immagine in modo da suddividerla in aree omogenee e rendere il confronto più semplice. Questa possibilità trova applicazione nell'ambito della *computer vision*
- *Sistemi OCR:* consiste nel riconoscere tramite supporto ottico caratteri stampati o manoscritti.
- *Information retrieval:* dato l'enorme sviluppo del Web sono necessari strumenti che garantiscano un efficiente reperimento delle informazioni di interesse. Questa tecnica è utilizzata anche per l'organizzazione e la ricerca dei libri e dei documenti nelle biblioteche.
- *Data Mining:* si utilizza in presenza di grandi quantità di dati al fine di estrarre informazioni significative con il giusto grado di sintesi.

4 L'elaborazione deve essere suddivisa in varie parti da assegnare a diversi elaboratori. L'esecuzione dei vari compiti deve essere sufficientemente autonoma da minimizzare lo scambio di informazioni tra le varie macchine.

Capitolo 2

Analisi del problema

Si supponga di avere un certo numero *N* di elementi ed una misura delle dissimilarità reciproche, supponiamo che queste misure siano organizzate in una matrice. Tale matrice, in genere, sarà simmetrica[5] con la diagonale nulla. Ciò che si vuole progettare è un algoritmo di clustering basato sulla densità che converga e sia efficace anche con matrici di grandi dimensioni. Il punto di partenza dello sviluppo è prendere in considerazione opportuni algoritmi basati sulla densità presenti in letteratura e procedere alla realizzazione di un nuovo algoritmo che rispetti le specifiche volute.

Gli algoritmi che verranno presi in considerazione sono due tra i più noti in letteratura: DBSCAN e OPTICS. In particolare quest'ultimo rappresenta una versione più avanzata del precedente. Si mostrerà che questi algoritmi sono stati sviluppati per lavorare nello spazio oggetto per cui necessiterebbero di una misura di distanza nello spazio di rappresentazione. Tuttavia è possibile, date le loro caratteristiche, adoperarli in uno spazio relazionale poiché, a differenza degli algoritmi oggetto, richiedono esclusivamente le distanze inter-pattern.

2.1 Il problema dell'alta dimensionalità

Nell'ambito della letteratura ricorre spesso il tema della significatività della distanza al fine di quantificare la diversità tra pattern. Si supponga di voler trovare il pattern più vicino ad un dato punto; si dimostra in termini matematici che, all'aumentare delle dimensioni (feature) e per dati distribuiti uniformemente, la distanza del punto più vicino tende ad essere uguale a quella del punto più lontano. Il teorema su cui si basa la precedente osservazione è il seguente:

5 Si vedrà in seguito che tale condizione non è strettamente necessaria.

Teorema:

Sia dato un punto q, scelto indipendentemente da tutti gli altri pattern X_i; sia $D_m(q,X)$ una variabile aleatoria che segue la distribuzione della distanza tra il punto q e i pattern; nell'ipotesi in cui sussiste la seguente espressione:

$$\lim_{m\to\infty} (var(D_m^p)/(E[D_m^p]^2)) = 0$$

dove p è l'esponente delle metrica di Minkovski usato (vale due per la distanza Euclidea), *var* rappresenta la deviazione standard ed E il valore medio. Sia M il numero di punti contenuti nell'ipersfera di raggio $(1+\varepsilon)r$ in cui r è la distanza dal pattern più vicino ed N il numero totale dei pattern;

allora per ogni $\varepsilon > 0$

$$\lim_{m\to\infty} (P[M = N]) = 1$$

Cioè la probabilità che nell'ipersfera siano contenuti tutti i pattern dell'insieme dato è pari ad 1.

□

Il senso della condizione sufficiente è il seguente: la deviazione standard deve aumentare più lentamente del valore atteso per la distanza del punto più vicino. Occorre notare che, se i vari pattern hanno molte coordinate uguali tra loro ($x_1=x_2=...=x_i$), l'effetto di mancanza significatività della distanza può non essere apprezzabile. Molte delle tecniche di clustering si basano sul presupposto che i punti di uno stesso cluster sono più vicini tra loro di quanto non lo siano punti di cluster diversi. Un modo di valutare se nell'insieme di dati ci sono dei cluster consiste nel costruire l'istogramma (i valori approssimano una funzione di densità di probabilità) delle distanze delle coppie (tutte oppure un campione se i dati sono molti). Nel caso in cui l'insieme di dati contenga diversi cluster il grafico presenta due picchi; uno si trova in prossimità della distanza media tra pattern e l'altro rappresenta la distanza tra pattern dello stesso cluster.

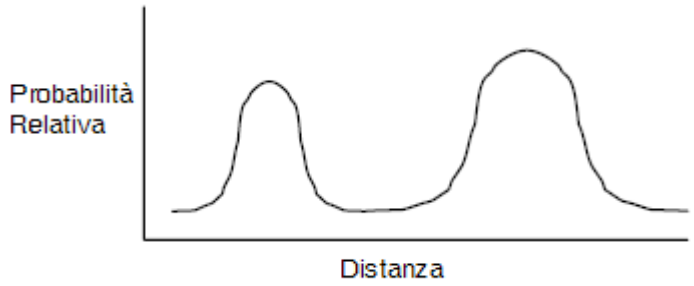

figura 6: Istogramma delle distanze reciproche fra pattern

Nel caso, invece, in cui non ci siano cluster, sarà presente un solo picco. All'aumentare

delle dimensioni (e nel caso in cui le condizioni del teorema precedente siano soddisfatte) i due picchi del grafico tendono ad avvicinarsi e, al limite, a confondersi fino a coincidere. In questo caso algoritmi come FCM sono impossibilitati ad effettuare un corretto clustering. Una possibile soluzione al problema consiste nell'utilizzo di algoritmi basati sulla densità operanti solo in base a valori di dissimilarità (espressa in forma relazionale).

Se, comunque, si utilizzano matrici di relazione molto grandi e si considerano le varie linee come coordinate di un singolo pattern, si osserva sperimentalmente che il calo di prestazioni di algoritmi come FCM non si evidenzia.

2.2 Algoritmi esistenti: DBSCAN

Il primo algoritmo che viene preso in considerazione è il DBSCAN (Density Based Spatial Clustering of Application with Noise) che introduce il concetto di densità e presenta ottime prestazioni. L'idea di fondo di DBSCAN è che, per appartenere ad un cluster, un pattern debba avere nelle sue vicinanze un numero minimo di punti; l'idea di vicinanza di DBSCAN presuppone solo una misura di distanza inter-pattern. Per la sua struttura, DBSCAN si presta bene allo studio di cluster reali con forme non necessariamente tondeggianti. Siano *minPts* ed ε due numeri reali, sia *D* l'insieme dei pattern; si danno le seguenti definizioni:

- ε-neighborhood di un punto *p*: è il vicinato del un punto *p* rispetto alla soglia ε ed è così definito: $N_\varepsilon(p)=\{q\in D|dist(p,q)\leq\varepsilon\}$
- Core-point: il punto *p* è detto core point se e solo se $Card(N_\varepsilon(p))\geq minPts$ cioè se nel suo vicinato ci sono almeno *minPts* punti
- Raggiungibilità diretta: un punto *p* è detto *direttamente raggiungibile per densità* (directly density-reachable) da un punto *q* rispetto a ε e *minPts* se:
  1) $p\in N_\varepsilon(q)$
  2) $|N_\varepsilon(q)|\geq minPts$
- Raggiungibilità: un punto *p* è detto *raggiungibile per densità* (density-reachable) da un punto *q* rispetto ε e *minPts* se c'è una successione di punti $p_1,....,p_n$, con $p_1=q$ e $p_n=p$, tale che p_{i+1} è raggiungibile direttamente da p_i
- Density connection: un punto *p* è *connesso per densità* (density connected) ad un punto *q* rispetto ad ε ed a *minPts* se e solo se esiste un punto *o* tale che *p* e *q* sono

raggiungibili per densità da *o*

Si noti come la proprietà di raggiungibilità diretta non sia simmetrica, a tale proposito si consideri anche l'esempio in figura 7.

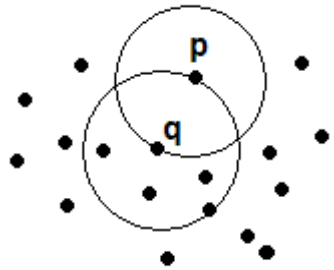

figura 7: *p* è direttamente raggiungibile per densità da *q*, mentre *q* non è direttamente raggiungibile per densità da *p*

La raggiungibilità indiretta è transitiva ma non simmetrica; mentre la connettività è simmetrica, riflessiva e transitiva. A questo punto è possibile introdurre la definizione di cluster:

Sia *D* un insieme di dati.

Un cluster *C* rispetto ad ε ed a *minPts* è un sottoinsieme non vuoto di *D* che soddisfa le seguenti condizioni:

- $\forall$ *p,q*: se $p \in C$ e *q* è raggiungibile per densità da *p* allora $q \in C$ "Massimalità"
- $\forall\ p, q \in C$: *p* è connesso per densità a *q* "Connettività"

Si definisce *rumore* l'insieme dei punti non appartenenti ad alcun cluster. Al fine di valutare la correttezza dell'algoritmo si introducono i lemmi seguenti.

Lemma 1: Dato un punto *p* del insieme di dati *D* con $|N_\varepsilon(p)| \geq minPts$, , l'insieme O = { *o* | *o* ∈ *D* , *o* raggiungibile per densità da *p* rispetto ad ε e *minPts*} è un cluster rispetto ad ε e *minPts*.

Lemma 2: Sia *C* un cluster rispetto ad ε e *minPts*, sia *p* un punto in *C* con $|N_\varepsilon(p)| \geq minPts$; in questo caso *C* coincide con l'insieme seguente

$O = \{ o \mid o \in D$, o raggiungibile per densità da p rispetto ad ε e $minPts\}$.

I lemmi precedenti suggeriscono il modo in cui operare per eseguire il clustering:

- Si scorre l'insieme dei punti non ancora elaborati alla ricerca dei core-point.
- Per ognuno di essi, si trovano tutti i punti raggiungibili per densità fino a formare il cluster.

La complessità di DBSCAN è data dalla scansione dell'insieme di dati a cui si deve aggiungere la 'region query' cioè la ricerca dei punti raggiungibili; nel caso peggiore tale ricerca si effettua su tutto il data set. Per limitare questo tempo di ricerca è possibile generare un indice di accesso ai dati, se tale indice è ad albero bilanciato, la complessità della ricerca è logaritmica. Globalmente si ha $O(n\ logn)$ il che significa che DBSCAN ha una discreta scalabilità.

Il clustering che ne consegue presenta alcuni limiti (quali ad esempio la difficoltà nel ricercare i parametri ottimali e la necessità di avere un valore di densità uguale per tutti i cluster) per ovviare ai quali è stato introdotto l'algoritmo OPTICS descritto in seguito.

2.3 Algoritmi esistenti: OPTICS

L'algoritmo OPTICS rappresenta una evoluzione del DBSCAN; ciò che l'algoritmo produce, però, non è un clustering esplicito, ma un ordinamento dei punti e l'aggiunta di informazioni che consentono di ricercare cluster di qualunque densità.

L'approccio di OPTICS è basato sulla considerazione che non tutti i dati reali sono caratterizzati da un parametro di densità globale; utilizzando un algoritmo che si basa su un parametro di densità globale, come DBSCAN, è possibile che non si riesca ad ottenere un partizionamento ottimale dei dati. Ad esempio, nella figura seguente si potrebbero distinguere *A,B,C* perdendo l'informazione sulla struttura interna di *C*, in alternativa si potrebbe fare in modo da distinguere *C1,C2,C3*, tuttavia gli altri punti sarebbero considerati come rumore.

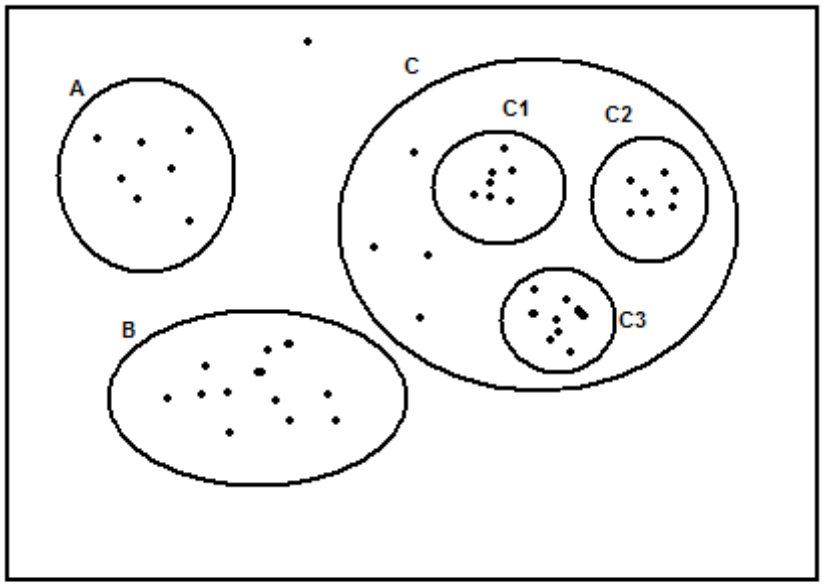

figura 8: Cluster di diversa densità nello stesso insieme di dati

Utilizzando, invece, un algoritmo gerarchico, si presenterebbe il problema del *single link*, cioè cluster collegati da una sottile striscia di punti non sarebbero distinguibili. La scelta di un algoritmo basato sulla densità e usato con parametri diversi pare poco efficiente (a causa della lentezza di elaborazione) e poco efficace (l'interpretazione dei risultati è molto complessa).

Tenendo conto delle informazioni precedenti, si introduce il concetto di ordinamento basato sulla densità (density-based cluster-ordering); si consideri la figura che segue:

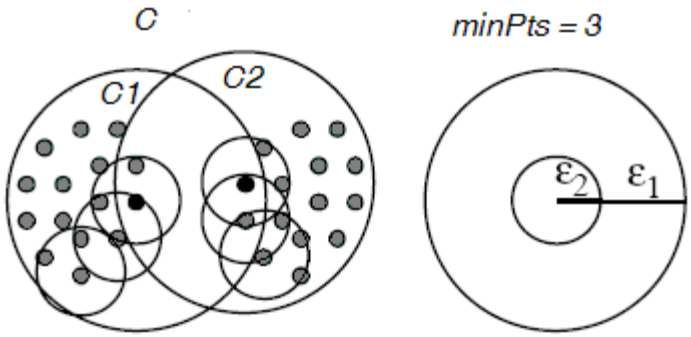

figura 9: Cluster ad alta densità sono contenuti in quelli a densità più scarsa

Per il medesimo valore di *minPts,* i cluster a densità più alta ε_2 sono contenuti in quelli a

densità più bassa ε_1; tenendo conto di ciò si può modificare DBSCAN in modo da processare cluster con densità via via minori. Concettualmente si deve effettuare un ordinamento in cui ogni elemento elaborato è di volta in volta quello densamente raggiungibile rispetto al minor ε in modo da elaborare prima i punti più vicini (cioè quelli nelle zone a densità maggiore). A tale proposito, oltre a quelle date per DBSCAN occorrono altre due definizioni (con *minPts* numero naturale ed ε numero reale).

- Core-distance di un punto *p*:

$$core\text{-}distance_{\varepsilon,\text{minPts}}(p) = \begin{cases} INFINITO & se\ Card(N_\varepsilon(p)) < minPts \\ minPts\text{-}distance(p) & altrimenti \end{cases}$$

in cui *minPts-distance*(*p*) indica il raggio del vicinato minimo di *p* (cioè il vicinato che contiene esattamente *minPts* punti). La core-distance rappresenta la più piccola distanza ε' tra *p* ed un oggetto di $N_\varepsilon(p)$ tale che *p* è ancora un core-object.

- Distanza di raggiungibilità (reachability-distance): siano *p* ed *o* due punti del data set *D*; sia $N_\varepsilon(o)$ il vicinato rispetto ad ε di *o*:

$$\text{reachability-distance}_{\varepsilon,\text{minPts}}(p,o) = \begin{cases} INFINITO & se\ Card(N_\varepsilon(o)) < minPts \\ max(core\text{-}distance_{\varepsilon,minPts}(o), d(o,p)) & \text{altrimenti} \end{cases}$$

Nella precedente espressione *d* rappresenta la distanza.

La reachability-distance e la più piccola distanza ε per cui *p* è direttamente raggiungibile per densità da *o* se *o* è core-object; in tal caso non è inferiore alla core-distance di *o* altrimenti nessun oggetto sarebbe direttamente raggiungibile (*o* non è più core object per distanze troppo piccole che rendono il numero dei vicini minore del limite *minPts*).

OPTICS ordina il database dei punti aggiungendo, per ognuno di essi, la core-distance e la reachability-distance. In questo modo si rende possibile l'estrazione dei cluster rispetto a varie densità.

Lo pseudo-codice che implementa OPTICS ha la seguente struttura:

```
OPTICS (SetOfObjects, e, MinPts, OrderedFile)
       OrderedFile.open();
       FOR i FROM 1 TO SetOfObjects.size DO
              Object := SetOfObjects.get(i);
              IF NOT Object.Processed THEN
                ExpandClusterOrder(SetOfObjects, Object, e,MinPts,OrderedFile)
       OrderedFile.close();
END; // OPTICS
```

Il ciclo riportato inizialmente apre il file in cui sarà scritto il risultato; tale risultato consiste in una lista ordinata di oggetti; questi oggetti sono gli elementi dell'insieme di dati in ingresso con l'aggiunta del valore della reachability-distance e della core-distance. Ogni punto non ancora processato è passato alla procedura **`ExpandClusterOrder`** riportata di seguito:

```
ExpandClusterOrder(SetOfObjects, Object, ε, MinPts,OrderedFile);
 neighbors := SetOfObjects.neighbors(Object, ε);
 Object.Processed := TRUE;
 Object.reachability_distance := UNDEFINED;
 Object.setCoreDistance(neighbors, ε, MinPts);
 OrderedFile.write(Object);
 IF Object.core_distance <> UNDEFINED THEN
       OrderSeeds.update(neighbors, Object);
       WHILE NOT OrderSeeds.empty() DO
              currentObject := OrderSeeds.next();
              neighbors:=SetOfObjects.neighbors(currentObject, ε);
              currentObject.Processed := TRUE;
              currentObject.setCoreDistance(neighbors, ε, MinPts);
              OrderedFile.write(currentObject);
              IF currentObject.core_distance <> UNDEFINED THEN
                     OrderSeeds.update(neighbors, currentObject);
END; // ExpandClusterOrder
```

Nel codice precedente la seed-list **`OrderSeeds`** rappresenta una lista di pattern non ancora elaborati ordinati secondo la loro reachability-distance rispetto al più vicino core-object direttamente raggiungibile.
La funzione compie i seguenti passi:

1. Calcola il vicinato (rispetto ad ε) del pattern.
2. Assegna alla reachability-distance del pattern il valore UNDEFINED (INFINITO,

in pratica un valore molto grande) e segna l'elemento come già elaborato.

3. Scrive il pattern e la relativa reachability-distance nel file di uscita.
4. Valuta se il punto è core-object rispetto alla distanza ε; se non lo è ritorna al ciclo principale altrimenti aggiorna la seed-list.
5. Entra in un ciclo nel quale per ogni elemento della seed-list (prendendo di volta in volta quello con la minore reachability-distance) ne calcola la core-distance; l'oggetto è quindi scritto nel file di uscita e segnato come elaborato. Se l'oggetto è un core-object, si aggiungono i suoi vicini alla seed-list.

Per quanto riguarda l'aggiornamento della seed-list, si utilizza la funzione che segue:

```
OrderSeeds::update(neighbors, CenterObject);
  c_dist := CenterObject.core_distance;
  FORALL Object FROM neighbors DO
       IF NOT Object.Processed THEN
             new_r_dist:=max(c_dist,CenterObject.dist(Object));
             IF Object.reachability_distance = UNDEFINED THEN
                   Object.reachability_distance := new_r_dist;
                   insert(Object, new_r_dist);
             ELSE // Object già OrderSeeds
                   IF new_r_dist < Object.reachability_distance THEN
                         Object.reachability_distance := new_r_dist;
                         decrease(Object, new_r_dist);
END; // OrderSeeds::update
```

La reachability-distance è determinata rispetto all'elemento `CenterObject`; oggetti non ancora presenti in lista sono semplicemente inseriti nell'ordine corretto; se l'elemento da inserire è già presente ne viene aggiornata la reachability-distance e la posizione in lista solo se il nuovo valore (calcolato rispetto al nuovo core object) è minore del precedente.
Gli autori di OPTICS hanno stimato che il tempo di esecuzione è 1.6 volte quello di DBSCAN e la complessità è stimabile come $O(n*query)$*;* dove query è il tempo di esecuzione della funzione che trova il vicinato di un punto. Tale query (cioè interrogazione), di norma, impone di scorrere tutto l'insieme dei dati, in questo caso la complessità globale è $O(n^2)$; se, invece, si fa uso di indici per l'accesso ai dati il valore precedente scende a $O(n \cdot \log n)$.
Dopo aver generato i risultati, per ottenere un clustering uguale a quello di DBSCAN, occorre semplicemente scorrere il file delle reachability-distance ed assegnare i punti

attigui con reachability-distance < ε' allo stesso cluster.
I risultati di OPTICS hanno una immediata interpretazione grafica, si supponga di elaborare un insieme composto da 3 cluster (anche ad alte dimensioni), creando un grafico dell'andamento delle reachability-distance si ottiene la figura seguente:

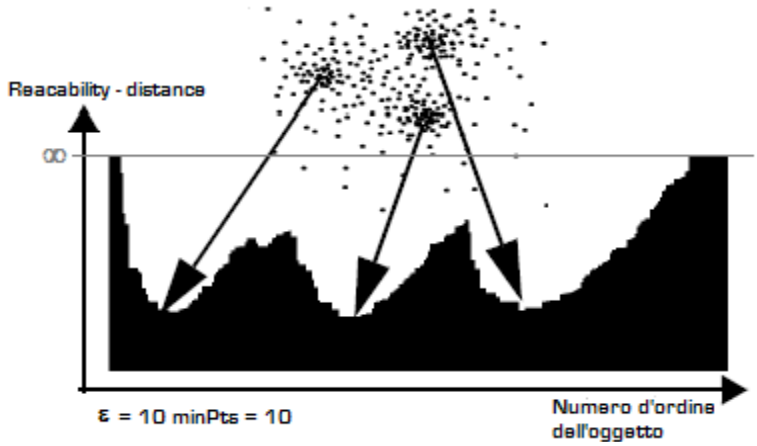

figura 10: Risultati dell'esecuzione di OPTICS su tre cluster

Come si vede i vari cluster sono rappresentati da avvallamenti del grafico, mentre i punti di rumore sono i punti con reachability-distance pari a ∞ (INDEFINITO, cioè un valore molto grande). Per quanto riguarda la scelta dei parametri e la loro influenza sui risultati dell'elaborazione vengono riportati di seguito degli esempi.
Si considerino, quindi, due valori $a > b$ a seconda che ε assuma l'uno o l'altro si hanno dei risultati di questo tipo:

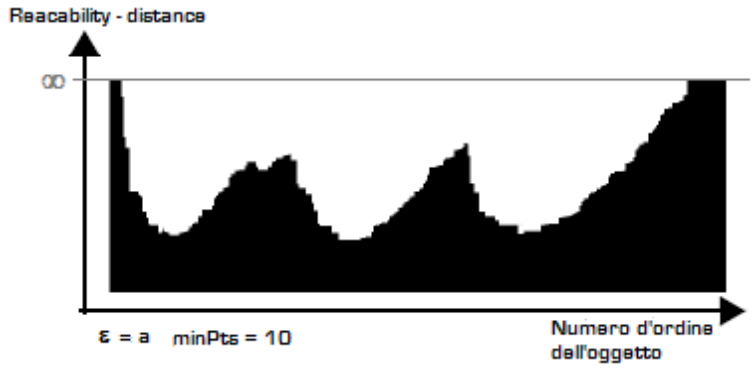

figura 11: Risultato di OPTICS con ε alto

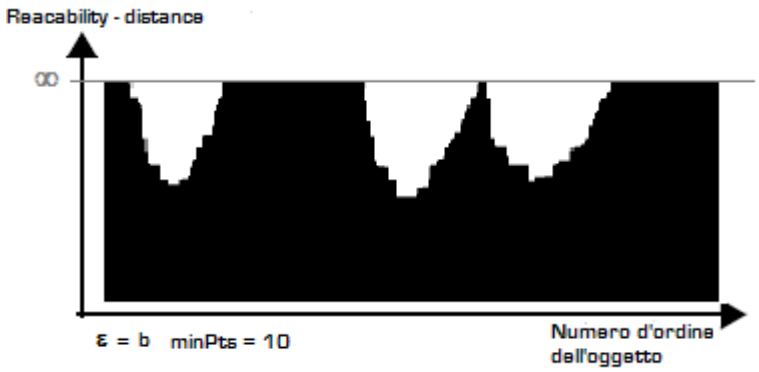

figura 12: Risultato di OPTICS con ε basso

Vale a dire che più piccolo è ε più ci saranno pattern che non appartengono ad alcun cluster (ci sarà, cioè, maggiore selezione).

Il parametro *minPts* influenza il numero di cluster che si individuano. Si supponga di variare *minPts* tra due valori *a* e *b* con $a < b$; i grafici del valore della reachability-distance sono i seguenti:

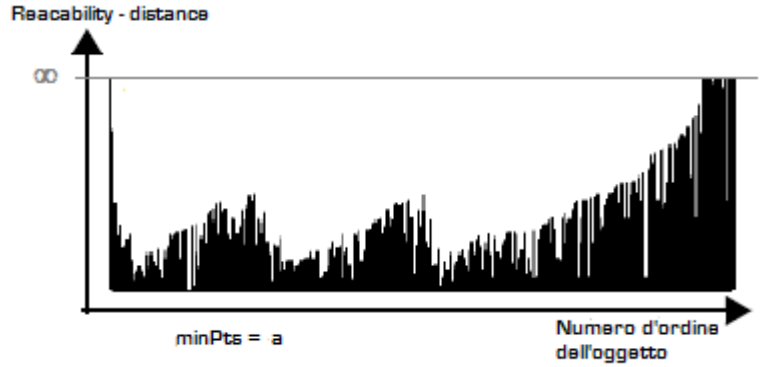

figura 13: Risultato di OPTICS con minPts basso

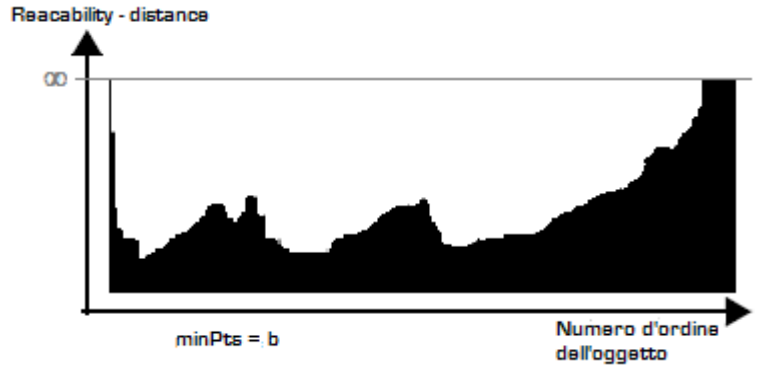

figura 14: Risultato di OPTICS con minPts alto

Se ne deduce che, abbassando il valore di *minPts,* si aumenta il numero di cluster trovati. Come si è visto la struttura di massima del clustering varia poco al variare dei parametri inoltre per valori prossimi il grafico non cambia per niente. Per la scelta del valore di ε, ci si può limitare a controllare che non sia troppo piccolo (altrimenti non si troverebbe

alcun cluster). Un approccio più rigoroso consiste nel porre ε pari al raggio di una ipersfera nello spazio delle feature contenente *minPts* pattern, supponendo la distribuzione di questi ultimi uniforme. Sia V_D il volume di una ipersfera nello spazio delle feature, il volume della ipersfera contenente *minPts* pattern vale:

$$V_S = \frac{V_D}{N} \times minPts$$

il volume di una ipersfera d-dimensionale vale:

$$V_S = \frac{\sqrt{\pi^d}}{\Gamma(\frac{d}{2}+1)} \times r^d$$

In cui Γ è la *funzione Gamma*[6].

Quindi, dato un d numero di feature ε vale:

$$\varepsilon = \sqrt[d]{\frac{V_D \times minPts \times \Gamma(\frac{d}{2}+1)}{N \times \sqrt{\pi^d}}}$$

Per la scelta del *minPts* gli autori suggeriscono valori pari a 10 o 20; prove sperimentali mostrano che per insiemi di dati complessi i valori migliori possono essere più bassi (2 o 5).

2.3.1 Individuazione dei cluster

A partire dai risultati di OPTICS occorre trovare i cluster; questi ultimi, in genere saranno annidati. L'idea di fondo consiste nell'analizzare il grafico dei risultati e cercare le regioni che rappresentano il potenziale inizio e fine dei cluster, combinandole in maniera opportuna. A tale proposito occorre interpretare i risultati di OPTICS in maniera formale e legare la definizione di cluster ad essi.

Ogni punto possiede un valore di reachability-distance (d'ora in poi *r*) che rappresenta la distanza del punto dai suoi predecessori; OPTICS, nella funzione **`ExpandClusterOrder`**

6 Si chiama funzione gamma la funzione $\Gamma : \mathbb{R}^+ \to \mathbb{R}^+$ definita da:

$$\Gamma(\alpha) = \int_0^{+\infty} x^{\alpha-1} e^{-x} dx$$

Si noti che tale integrale non è analiticamente calcolabile. Nei casi pratici, per effettuare il calcolo, si utilizzano tavole numeriche. Si noti anche che per valori interi l'integrale diventa: Γ(*n*)=(*n*-1)!

elabora di volta in volta il punto più vicino al precedente. Tenendo conto di ciò si può considerare l'esempio in figura 15.

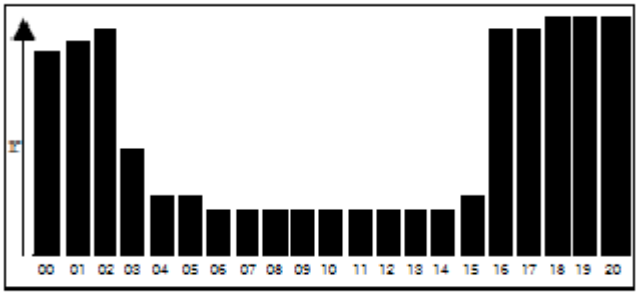

figura 15: Esempio di risultato di OPTICS

Il punto di inizio del cluster è, in genere, l'ultimo che ha un valore alto di *r* (qui il punto 3 compreso), mentre l'ultimo è l'ultimo che ha *r* basso. Spesso i risultati reali hanno cluster con inizio e fine meno ripidi. Per gestire i diversi gradi di ripidità occorre inserire l'ulteriore parametro ξ; inoltre sono date le seguenti definizioni:

ξ-steep points:

Un punto si chiama ξ-steep upward se e solo se è più basso dello ξ% del successivo:

$$UpPoint_{\xi}(p) \Leftrightarrow r(p) \leq r(p+1) \times (1-\xi)$$

Un punto si chiama ξ-steep downward se e solo se il suo successore è dello ξ% più basso:

$$DownPoint_{\xi}(p) \Leftrightarrow r(p) \times (1-\xi) \geq r(p+1)$$

Dove *r*(*p*) è la reachability-distance di *p*.

ξ-steep areas:

Un intervallo $I = [s,e]$ si chiama ξ-steep upward area se e solo se soddisfa le condizioni che seguono:

- *s* è ξ-steep upward point
- *e* è ξ-steep upward point
- Ogni punto è alto almeno come il precedente: $\forall x,\ s < x \leq e : r(x) \geq r(x-1)$
- Non contiene più di *minPts* punti consecutivi che non sono ξ-steep upward: $\forall [\bar{s}, \bar{e}] \subseteq [s, e] : ((\forall x \in [\bar{s}, \bar{e}] : \neg\, UpPoint_{\xi}(x)) \Rightarrow \bar{e} - \bar{s} < minPts)$
- I è massimale: $\forall J : (I \subseteq J, UpArea_{\xi}(J) \Rightarrow I = J)$

Un intervallo $I = [s,e]$ si chiama ξ-steep downward area se e solo se soddisfa le condizioni che seguono:

- s è ξ-steep downward point
- e è ξ-steep downward point
- $\forall x,\ s<x\leq e:\ r(x)\leq r(x-1)$
- Non contiene più di *minPts* punti consecutivi che non sono ξ-steep downward:
  $\forall [\bar{s},\bar{e}] \subseteq [s,e]:\ ((\forall x \in [\bar{s},\bar{e}]:\ \neg\ DownPoint_{\xi}(x)) \Rightarrow \bar{e}-\bar{s} < minPts)$
- I è massimale: $\forall J:\ (I \subseteq J,\ DownArea_{\xi}(J) \Rightarrow I = J)$

Si noti come una ξ-steep area non debba contenere più di *minPts* punti non ξ-steep (ciò significa che qualora ci siano più di *minPts* punti consecutivi che non presentano gradini implicitamente i punti successivi apparterranno ad un altro cluster). A questo punto si può legare la definizione di cluster alle definizioni precedenti:

<u>ξ-cluster:</u>

Un intervallo $C = [s,e] \subseteq [1,n]$ è chiamato ξ-cluster se e solo se soddisfa le seguenti 4 condizioni:

$cluster_{\xi}(C) \Leftrightarrow \quad \exists D = [s_D, e_D]\ ,\quad \exists U = [s_U, e_U]$ con:

1. $DownArea_{\xi}(D)\ \wedge\ s \in D$
2. $UpArea_{\xi}(U)\ \wedge\ e \in U$
3. a) $e\text{-}s \geq minPts$
   b) $\forall x,\ s_D < x < e_U:\ (r(x) \leq min(r(s_D), r(e_U+1)) \times (1-\xi))$
4. $(s,e)=$

$$\begin{cases} (max\{x \in D \mid r(x) > r(e_U+1)\}, e_U) & \text{se } r(s_D) \times (1\text{-}\xi) \geq r(e_U+1) \\ (s_D, min\{x \in U \mid r(x) < r(s_D)\}) & \text{se } r(e_U+1) \times (1\text{-}\xi) \geq r(s_D) \\ (s_D, e_U) & \text{altrimenti} \end{cases}$$

Le condizioni 1 e 2 stabiliscono che il cluster inizia in una ξ-steep downward area e finisce in una ξ-steep upward area. In un cluster ci sono almeno *minPts* punti (condizione 3a). Inoltre i punti appartenenti al cluster devono essere dello ξ% più in basso del primo

punto dell'area ξ-steep downward e dell'ultimo punto dell'area ξ-steep upward area. La condizione 4 stabilisce che tra i valori di inizio e fine ci deve essere una differenza di altezza minore dello ξ%.

L'algoritmo che estrae i cluster (tutti) dal vettore delle reachability-distance verrà descritto in due versioni, la prima è quella concettualmente più semplice, mentre la seconda è più efficiente.

Essenzialmente ciò che si fa consiste nello scorrere il vettore risultato di OPTICS, memorizzando le aree ξ-steep downward e cercando di combinarle con le successive aree ξ-steep upward. L'approccio più semplice consiste nelle operazioni che seguono:

1. Porre *index* = 0
2. Finché *index* < *n* ripetere le seguenti istruzioni:
3. Se una area steep-downward inizia in *index* aggiungerla all'insieme delle aree downward e proseguire.
4. Se una nuova area steep-upward inizia in *index* combinarla con tutte le aree steep-downward trovate e, per ogni coppia trovata, verificare le condizioni ξ-cluster; se sono soddisfatte calcolare inizio e fine del cluster in base alla condizione 4.
5. Incrementare *index* e tornare al punto 2.

Accade che molte delle combinazioni trovate non rispettano le condizioni e sono scartate; per limitare il numero delle potenziali aree di inizio cluster si debbono operare alcune trasformazioni sulle condizioni descritte in precedenza. Si consideri la condizione di cui al punto 3b:

$$\text{3b:} \quad \forall\, x,\ s_D < x < e_U : (r(x) \le min(r(s_D), r(e_U+1)) \times (1-\xi))$$

tale espressione si può riscrivere nel seguente modo:

$$\text{(sc1)} \quad \forall\, x,\ s_D < x < e_U : (r(x) \le r(s_D) \times (1-\xi))$$
$$\text{(sc2)} \quad \forall\, x,\ s_D < x < e_U : (r(x) \le r(e_U+1) \times (1-\xi))$$

Le due condizioni si possono ancora trasformare[7]:

$$\text{(sc1*)} \quad max\{x \mid s_D < x < e_U\} \leq r(s_D) \times (1-\xi)$$
$$\text{(sc2*)} \quad max\{x \mid s_D < x < e_U\} \leq r(e_U+1) \times (1-\xi)$$

Per sfruttare la trasformazione si introduce il concetto di *mib* (maximum in between) che rappresenta il massimo valore nell'intervallo tra un certo punto e l'indice corrente. Occorrerà quindi tenere traccia del *mib* di ogni area steep-downward (rispetto alla fine dell'area e all'indice corrente) e del *mib* globale che rappresenta il massimo valore dell'intervallo tra la fine dell'ultima area steep (upward o downward) e l'indice corrente. L'algoritmo che deriva dalle osservazioni precedenti è riportato di seguito:

```
setOfSteepDownAreas = vuoto
setOfClusters = vuoto
index = 0, mib = 0
WHILE(index<n)
  mib = max(mib,r(index)) // r(index) è la reachability-distance
                                dell'elemento index
  IF(inizio area steep down D in index)
        aggiornare i valori mib e filtraggio delle aree downward (*)
        D.mib = 0
        aggiungere D all'insieme delle down area
        index = fine D + 1, mib=r(index)
  ELSE IF(inizio area steep upward U in index)
           aggiornare i valori mib e filtraggio delle aree downward
           index = fine U + 1, mib=r(index)
           FOR EACH D dell'insieme delle aree downward
               IF(combinazione D + U è cluster(**) e soddisfa le condizioni 1,2,3a)
                     calcolare [s,e] e aggiungerlo all'insieme dei cluster
       ELSE index = index + 1
RETURN clusters
```

Per filtraggio delle aree ξ-steep-downward si intende l'eliminazione di quelle il cui inizio

7 L'articolo che presenta OPTICS propone la condizione 3b in questo modo:

$\forall x,\ s_D < x < e_U : (r(x) \leq min(r(s_D), r(e_U)) \times (1-\xi))$

aggiungendo il termine +1: si verifica la condizione sul punto più alto che è il primo dopo l'area steep-upward, cioè il primo non appartenente al cluster. Se non si fosse fatto questo, l'algoritmo non sarebbe stato in grado di distinguere cluster che terminano bruscamente (da *r* basso a infinito).

(reachability distance) moltiplicato per (1-ξ) è minore del *mib* globale; così facendo si riduce il numero di cluster possibili e si soddisfa al contempo (sc1*). Al punto (**) viene confrontato il primo valore successivo alla area upward moltiplicato per (1-ξ) con il *mib* della area downward soddisfacendo (sc2*). La procedura di estrazione dei cluster ha complessità lineare $O(n)$, come, del resto, è confermato dai dati sperimentali forniti dagli autori. La figura seguente mostra i tempi di estrazione dei cluster relativi ad un insieme di dati reali costituito da fotogrammi televisivi rappresentati da un istogramma del colore a 64-dimensioni.

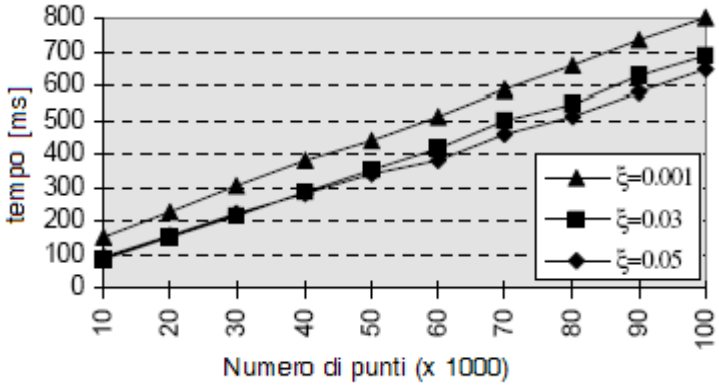

figura 16: Durata dell'esecuzione della routine di estrazione dei cluster per differenti parametri

2.4 Fondamenti teorici del nuovo algoritmo

Le caratteristiche di OPTICS lo rendono applicabile su dati relazionali, caratterizzati dalle dissimilarità di ogni punto rispetto a tutti gli altri elementi dell'insieme. Supponiamo, ad esempio, di possedere il seguente insieme di punti in $\mathbb{R}^2$:

$XY=\{(1,1),(1,2),(2,1),(2,2),(3,3)\}$

Un modo sintetico per estrarre la matrice di dissimilarità da questi punti è quello di calcolare le distanze[8] reciproche tra i vari punti ottenendo la matrice seguente:

8 In questo caso si è usata la distanza cosiddetta di Manhattan definita come segue: dati due vettori *x* ed *y* la loro distanza di Manhattan è: *sum*(a*bs*(*x*-*y*))
in alternativa si usa la distanza o norma Euclidea o la distanza di Mahalanobis.

$$R = \begin{pmatrix} 0 & 1 & 1 & 2 & 4 \\ 1 & 0 & 2 & 1 & 3 \\ 1 & 2 & 0 & 1 & 3 \\ 2 & 1 & 1 & 0 & 2 \\ 4 & 3 & 3 & 2 & 0 \end{pmatrix}$$

Un'altra caratteristica interessante di OPTICS è che non ha problemi di convergenza ed i valori di dissimilarità non devono necessariamente rappresentare delle distanze cioè lo spazio di rappresentazione non deve essere necessariamente uno spazio metrico[9] (per altri algoritmi, come si vedrà in seguito, questo è necessario). Occorre, a questo punto adattare le definizioni date per il DBSCAN e OPTICS al caso relazionale. Supponiamo di avere un matrice simmetrica[10] e quadrata R di dimensioni $N \times N$, dati due pattern p e q la loro dissimilarità è data dall'elemento della matrice di coordinate p e q: $R(p,q) = R(q,p)$. Valgono le seguenti definizioni:

ε-neighborhood di un punto *p*: $N_\varepsilon(p) = \{q \in R \mid R(p,q) \leq \varepsilon\}$

Core-point *p*: p è un core point se e solo se $Card(N_\varepsilon(p)) \geq minPts$

minPts-distance: la minima distanza d per cui nel d-neighborhood ci sono almeno *minPts* elementi.

Core-distance: dato l'elemento p;

$$core\text{-}distance_{\varepsilon,minPts}(p) = \begin{cases} INDEFINITA & se\ Card(N_\varepsilon(p)) < minPts \\ minPts\text{-}distance & \text{altrimenti} \end{cases}$$

9 Si definisce spazio metrico un insieme I a valori in $\mathbb{R}$ tale che è definita una funzione di distanza d da $I \times I$ in $\mathbb{R}_0^+$ che rispetta le seguenti condizioni:

- $d(a,b) = 0 \Leftrightarrow a = b$
- $d(a,b) = d(b,a) \quad \forall a, b \in I$
- $d(a,c) \leq d(a,b) + d(b,c)$

10 Se la matrice non è simmetrica può essere trasformata sostituendo a due elementi simmetrici diversi ad esempio il loro valore medio e ponendo gli zeri sulla diagonale.

Reachability-distance del punto *p* rispetto al punto *o*:

$$reachability\text{-}distance_{\varepsilon,minPts}(p,o) = \begin{cases} INDEFINITA\ se\ \ Card(N_{\varepsilon}(o)) < \ minPts \\ max(core\text{-}distance_{\varepsilon,minPts}(o), R(p,o)) \end{cases}$$

Come si evince dalle definizioni fornite, in ingresso sono presenti due parametri: *minPts* ed ε. Si noti che la struttura della funzione di calcolo delle reachability-distance risulta identica al caso di OPTICS (tenendo conto, ovviamente, delle nuove definizioni). Come si è detto nell'introduzione, uno degli inconvenienti maggiori nell'applicazione di un algoritmo gerarchico consiste nella difficoltà di costruire il cosiddetto dendrogramma; si noti, comunque, che OPTICS genera un ordinamento dei punti, rendendo attigui quelli dello stesso cluster, in questo modo per costruire il diagramma basterà memorizzare in un albero i punti di inizio e fine di ogni cluster (rispetto al nuovo ordinamento) a partire da quelli più piccoli fino a comprenderli tutti sotto una radice comune che rappresenta l'intero insieme di dati. Per quanto riguarda le definizioni riguardanti l'estrazione dei cluster, esse sono identiche a quelle di OPTICS e non vengono riportate.

Capitolo 3

Progettazione di ROPTICS

Nel presente del capitolo sarà descritta in dettaglio l'algoritmo Relational OPTICS (ROPTICS), oggetto della tesi. Si tratta di un algoritmo relazionale derivato da OPTICS, con buone caratteristiche sia in termini di convergenza che di analisi per alta dimensionalità.

3.1 Operazioni svolte

Le operazioni che seguono servono a generare i valori di reachability-distance.

1. Ordinamento preliminare: si scorre l'insieme di dati e si segnano i punti che rappresentano core-object.
2. Si prende l'elemento successivo non ancora processato dando la priorità ai core-objects; se sono finiti gli elementi, si termina.
3. Si pone la reachability-distance dell'oggetto a INFINITO; si segna l'oggetto come processato e se ne calcola la core-distance rispetto a ε e *minPts* . L'elemento con la relativa reachability-distance viene scritto nel file di uscita.
4. Se la core-distance è INDEFINITA si torna al passo 2 altrimenti si prosegue.
5. Si trovano tutti i membri del ε-neighborhood dell'elemento e se ne calcola la reachability-distance rispetto all'elemento in questione. Si inseriscono i membri in un insieme chiamato seed-list con il corrispondente valore di reachability-distance; se alcuni di essi sono già presenti se ne aggiorna il precedente valore di reachability-distance se la nuova è minore (mantenendo la lista ordinata per valori crescenti).
6. Per ogni elemento della seed-list si effettuano i seguenti passi:

a. Si prende il minore (reachability-distance più piccola)
b. Lo si segna come elaborato; se la core-distance non è INDEFINITA si aggiorna la seed-list con i suoi vicini (i membri del neighborhood) e relativa reachability-distance
c. L'elemento viene scritto sul file di uscita

7. Si torna al punto 2

La differenza rispetto ad OPTICS riguarda l'ordinamento preliminare; si è visto che, in mancanza di questa operazione, l'algoritmo può scambiare un border-object per rumore se questo è elaborato prima del core-object che lo contiene nel suo vicinato, in quanto per un border-object la core-distance vale INFINITO; nella implementazione esistente di OPTICS nello spazio oggetto, una volta terminati i punti contenuti nella seed-list si passa a considerare il punto non ancora elaborato più vicino all'origine, tale approccio non è possibile nel caso di ROPTICS. Si noti, comunque, che nel seguito ε viene posto sempre ad infinito per fare in modo che ogni pattern venga assegnato ad almeno un cluster; in questo modo l'ordinamento iniziale è inutile e non viene effettuato (poiché non ci sono border-object).

La procedura che estrae i cluster annidati si basa sulle definizioni date per OPTICS; per quanto riguarda il calcolo dei punti estremi del cluster si segue il seguente schema, in accordo alla definizione di ξ-cluster:

8. Per tutte le coppie valide di aree ξ-steep-downward/upward desunte dal vettore delle reachability-distance:
    a. Se lo ξ% della reachability-distance del punto di inizio dell'area steep-downward è maggiore della reachability-distance della fine dell'area steep-upward allora il cluster inizia nel punto corrispondente al valore più basso di reachability-distance tra quelli con valore maggiore dell'ultimo punto dell'area steep-upward.
    b. Se lo ξ% della reachability-distance del punto di fine della area steep-upward è maggiore della reachability-distance dell'inizio dell'area steep-downward allora il cluster finisce nel punto corrispondente al valore più alto di reachability-distance tra quelli con valore minore del primo punto della steep-downward.
    c. Negli altri casi il cluster inizia nel primo punto della steep-downward e

finisce nell'ultimo punto della steep-upward.

d. Si inserisce il cluster nell'albero generico.

Ogni volta che si identifica un cluster, questo deve essere memorizzato in modo da tenere traccia delle relazioni con gli altri cluster (la procedura di estrazione produce cluster annidati); a tale scopo si introduce la struttura ad albero generico. L'albero generico è rappresentabile come un insieme di elementi, detti *nodi* che possiedono un campo informativo e due puntatori[11], uno ai *nodi figli* ed uno ai *nodi fratelli*; se un nodo non ha figli viene detto *foglia*. In questo modo si è in grado di memorizzare efficacemente la struttura gerarchica. Supponiamo che il campo informativo contenga il numero[12] di inizio e fine del cluster; per il modo in cui la procedura **`ExtractCluster`** cerca i cluster (prima quelli interni), c'è la certezza che il cluster trovato non sia contenuto in alcun altro cluster già presente nell'albero. Volendo inserire un nuovo cluster si procede come segue:

9. Si scorrono i fratelli della radice alla ricerca di un elemento che abbia il numero di fine cluster minore dell'elemento da inserire ed il numero di inizio maggiore.
10. Se non si trova, si inserisce il cluster come fratello della radice nell'ordine giusto (non ci sono sovrapposizioni).
11. Se, invece, viene trovato, questo diviene figlio del nuovo elemento che ne prenderà il posto originario. Se gli elementi che soddisfano la condizione sono più di uno, diverranno tutti figli del nuovo elemento.

Data la struttura ad albero, si vuole, ora, estrarre un certo numero predefinito di cluster. Anzitutto occorre introdurre una misura della qualità di un cluster. Tale misura sarà usata per ordinare i cluster in basso nella gerarchia (foglie dell'albero), in modo da poter scegliere i migliori che vengono via via espansi affinché ogni pattern appartenga ad un cluster. Si definisce indice di qualità del cluster la seguente quantità:

$$q = \frac{(end - begin)^2}{\sum_{i=begin}^{end} reachability_i}$$

11 Il puntatore è una variabile contenente un indirizzo di memoria; in questo caso i puntatori formano i collegamenti tra i nodi dell'albero presenti in memoria.

12 Tale numero si riferisce, ovviamente, all'ordine stabilito dal ciclo in comune con OPTICS.

Questa quantità tiene conto sia del valore medio delle reachability-distance, che deve essere basso per indicare una densità alta, sia del numero di punti del cluster. L'algoritmo che trova *N* cluster ha, quindi, una struttura simile alla seguente:

12. Si ordinano i nodi foglia in una lista ordinata in base a *q*.
13. Vengono segnati gli *N* elementi dell'albero corrispondenti ai primi della lista, se non se ne hanno abbastanza l'algoritmo termina.
14. Si espandono i fratelli segnati fino ad essere attigui (eventualmente inglobando i fratelli che non hanno foglie segnate) e fino a ricoprire tutti gli elementi del nodo padre, a questo punto il/i fratelli prendono il posto del padre.
15. Si ripete il punto 14 finché i fratelli della radice non sono tutti foglie e segnati.

L'algoritmo termina qui. È importante, ora, soffermarsi sulla scelta dei parametri. Come si è detto il valore ottimale per ε, nel caso in cui si voglia partizionare tutto il data set, è pari ad INFINITO; per quanto concerne il *minPts* occorre fare alcune precisazioni. Generalmente, anche nel caso di OPTICS, tale valore è posto uguale a 10 o 20, spesso non è determinante tranne nel caso in cui il data set sia molto piccolo o di forma particolare. Una formula empirica che negli esperimenti si è rivelata utile è la seguente:

$$minPts = 0.4 \times \frac{N}{C}$$

Il senso della precedente formula consiste nel fatto che la funzione di estrazione degli *N* cluster a partire dall'albero è approssimativa; tenendo alto il *minPts* vengono filtrati i cluster piccoli che possono rendere inefficiente la precedente estrazione. Nel caso in cui i dati siano particolarmente difficili (ad esempio cluster con numero di punti molto differenti tra loro) può rendersi necessario adottare un valore basso (pari, talvolta a 2).
Per quanto riguarda lo ξ, è importante che sia sufficientemente basso ad esempio 10^{-5}.

3.2 Ulteriori aspetti rilevanti

Nei data set reali può esserci parziale sovrapposizione tra i cluster. I punti facenti parte di vari cluster hanno basse dissimilarità verso i punti di quei cluster.
Supponiamo di possedere un insieme di dati prodotto a partire dalla misura di 4 caratteristiche di alcuni esemplari di iris.

Caratteristiche dell'insieme di dati:
Numero pattern: 150
Numero Classi: 3 (50 pattern per classe, ogni classe corrisponde ad una varietà di iris:Iris Setosa, Iris Versicolour, Iris Virginica
Numero feature: 4 (lunghezza del sepalo in cm, larghezza del sepalo in cm, lunghezza del petalo in cm,larghezza del petalo in cm).

Due dei tre cluster non sono linearmente separabili. Se si costruisce artificialmente la matrice relazionale con la distanza non si ottengono i risultati voluti; nel seguito si utilizzerà una matrice prodotta da un particolare algoritmo che estrae la somiglianza dai dati (descritto nell'articolo di M. G. Cimino, B. Lazzerini, F. Marcelloni: *A Novel Approach to Fuzzy Clustering based on a Dissimilarity Relation extracted from Data using a TS System*). Supponiamo che i punti siano ordinati per appartenenza ai tre cluster; assegnando ad ogni valore una gradazione di grigio, tramite un software come Matlab™ si ottiene una rappresentazione del tipo in figura 17; le caselle scure si trovano in concomitanza di punti simili mentre le zone chiare rappresentano le relazioni tra punti di cluster diversi (ovviamente i punti sono ordinati in base al cluster di appartenenza).

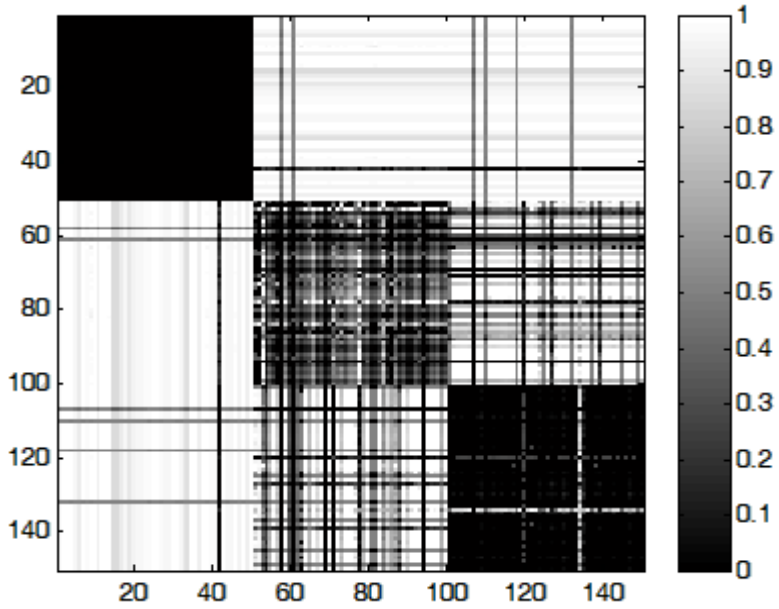

figura 17: Rappresentazione della matrice di dissimilarità dell'insieme di dati Iris; i punti sono ordinati per cluster di appartenenza. Le caselle scure sulla diagonale rappresentano i 3 cluster del data set

Tenendo conto della rappresentazione degli oggetti in uno spazio relazionale, si può

innanzitutto osservare che i punti con bassa norma[13] hanno una dissimilarità media bassa, e pertanto sono degli oggetti "ibridi", simili a molti oggetti del data set, anche appartenenti a classi diverse. Sono cioè dei core-object il cui vicinato contiene molti punti di altri cluster. Nella figura precedente si nota che l'elemento 42 è poco diverso da tutti gli altri e che esistono diversi elementi a comune tra i cluster 2 e 3. Una possibile soluzione a questo problema potrebbe consistere nell'elaborare per ultimi i punti che hanno una norma inferiore ad un certo limite ed assegnarli al cluster rispetto ai punti del quale posseggono dissimilarità minore. La dissimilarità di un punto *p* da un cluster si può calcolare facendo la media delle dissimilarità di *p* dai punti che compongono il cluster.

Si noti che la soluzione presentata non sempre è corretta; difatti può capitare che alcuni punti abbiano basse dissimilarità verso due cluster ma mediamente la dissimilarità sia minore verso il cluster a cui non appartengono (ad esempio i punti 61,69 e 94 120 in figura 17 presentano caselle mediamente più scure in presenza dei punti del cluster a cui non appartengono mentre le caselle che corrispondono alla dissimilarità verso i punti del cluster di appartenenza sono mediamente più chiare). La soluzione che è stata adottata consiste nel considerare i punti a norma bassa come border-object ed assegnarli al cluster a cui appartiene il pattern che li contiene nel proprio vicinato che è stato elaborato per primo. Il valore limite della norma, sotto il quale il punto viene considerato come 'ibrido', è considerato come ulteriore parametro in ingresso, che sarà chiamato ν.

Tornando all'esempio precedente, posto ν = 85, ROPTICS identifica i tre cluster, pur con un 14% di errori.

Nell'esempio di figura 17, la matrice è stata ottenuta attraverso un sistema di estrazione automatica di somiglianza, guidato da un sottoinsieme estremamente ridotto di somiglianze di training pari al 10% dell'insieme. È possibile, quindi, che vi siano delle somiglianze non realistiche.

Se, a partire dallo spazio oggetto, otteniamo la matrice di dissimilarità tramite distanza

13 Si dice *norma vettoriale* e si indica con $||x||$ una funzione definita nello spazio vettoriale $\mathbb{C}^N$,a valori reali non negativi, che verifica le seguenti condizioni:

$||x||=0 \Leftrightarrow x=0$ cioè se e solo se x è il vettore nullo la sua norma è zero

$||\alpha x||=|\alpha|||x|| \quad \forall x \in \mathbb{C}^N, \forall \alpha \in \mathbb{C}$

$||x+y|| \leq ||x||+||y|| \quad \forall x, y \in \mathbb{C}^N$

In $\mathbb{C}^N$ si possono definire norme in modo arbitrario; tra le più usate si ricorda la *norma 1:*

$$||x||_1 = \sum_{i=1}^{N} |x_i|$$

Euclidea si ha una matrice raffigurata in figura 18.

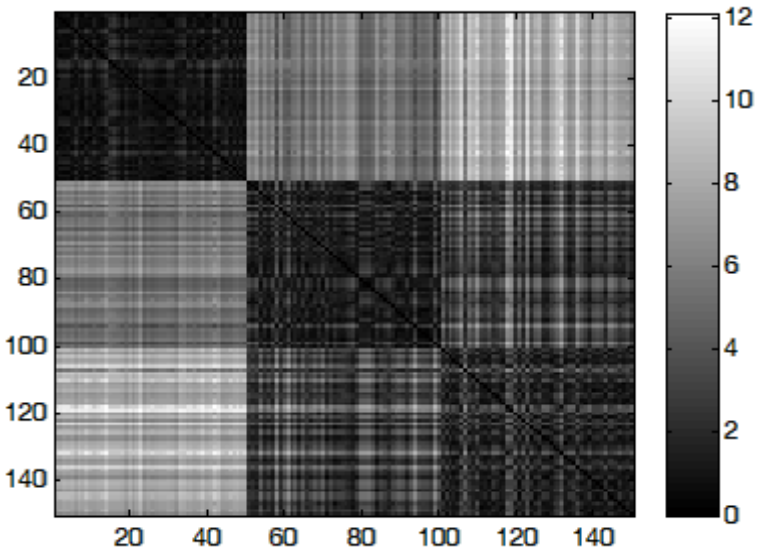

figura 18: Rappresentazione della matrice di dissimilarità del insieme di dati Iris ottenuta tramite distanza Euclidea

Eseguendo ROPTICS con minPts=5 e ν = 80, si ha l'8% di errori.

È chiaro che per stimare ν si deve necessariamente avere una conoscenza a priori dei dati e della forma della matrice; in generale la ricerca di tale valore non è banale.

3.3 Complessità asintotica

Nel seguito del paragrafo si farà riferimento ai punti di cui al paragrafo 3.1.

Anzitutto occorre valutare la complessità della procedura di ordinamento preliminare; si noti, comunque, che nel caso in cui ε valga ∞ la procedura non è eseguita. La procedura consiste nell'eseguire M volte (con M numero di pattern) una certa funzione `is_core` che valuta se un oggetto è core-object oppure no; tale funzione controlla che ogni oggetto abbia almeno *minPts* vicini, e quindi, nel caso peggiore, devo scorrere tutti i valori di dissimilarità (M cicli). Nel complesso si ha $O(M^2)$.

Il numero di iterazioni di ROPTICS è determinato essenzialmente dalla struttura simile ad OPTICS. In tale procedura vengono presi in esame tutti i punti e per ognuno occorre calcolarne il vicinato; questa operazione consiste nello scorrere la linea della matrice corrispondente. Si ottiene quindi $O(M^2)$. Per quanto concerne l'estrazione dei cluster annidati, si deve operare su di un vettore. In genere questa operazione comporta un tempo trascurabile ed ai fini del calcolo della complessità può essere trascurata (poche frazioni di secondo contro ore di elaborazione). L'operazione consiste nel valutare se un

punto è uno steep-point ed eventualmente aggiungerlo in una lista, ed ha complessità lineare; la combinazione tra le *D* aree steep-downward e le *U* aree steep-upward è quadratica, infatti ogni steep-upward viene confrontata con ogni steep-downward (nel caso peggiore); si ha quindi $O(DU)$. Si noti che, in ogni caso, molte delle ξ-steep down area vengono filtrate, in tal modo la procedura ha complessità lineare.
Anche la trasformazione dell'albero è molto veloce; le operazioni fatte sono le seguenti:

- Visita dell'albero: per visita dell'albero si intende, qui, l'esame di tutti i nodi; la complessità è lineare.
- Per ogni foglia dell'albero si calcola l'indice di qualità. La complessità del calcolo del coefficiente è lineare. Tenendo conto che le foglie dell'albero rappresentano cluster non sovrapposti, sia *C* il numero di cluster la complessità massima dell'operazione è $O(C)$
- La ricerca dei migliori *C* cluster è lineare.
- La valutazione della complessità della funzione che estrae gli *C* cluster finali è molto complessa. Comunque è inferiore alla complessità quadratica (sperimentalmente richiede frazioni di secondo rispetto ai diversi minuti richiesti dalla funzione di ricerca delle reachability-distances). Si supponga di avere *C* foglie dell'albero segnate, per trovarle si scorre tutto l'albero; trovatane una la si espande fino a che diventi attigua ai fratelli (modificando i valori di inizio e fine del cluster). In seguito i fratelli attigui vengono modificati in modo da comprendere tutti i pattern del nodo padre e, infine, prendono il suo posto. Al massimo si calcola la funzione *N* volte (*N* numero di nodi) per portare su di un livello i nodi segnati; successivamente si esegue *R* volte (con *R* pari al numero di nodi foglia e del penultimo livello rimasti) finché non si hanno solo foglie della radice, cioè al massimo *H* volte con *H* numero di livelli. Poiché *R* diminuisce ogni volta la complessità non raggiunge quella quadratica (si veda a riguardo anche il codice sorgente).

Si può concludere che ROPTICS ha una complessità quadratica che consiste essenzialmente nel ciclo a comune con OPTICS cioè $O(M^2)$ dove *M* è il numero di pattern. Nel caso in cui vengano generati degli indici di accesso spaziali ad albero, la complessità della ricerca dei neighbors diventa logaritmica; in tal modo si scende ad $O(M\log M)$.

Capitolo 4

Sviluppo dell'applicazione

Viene riportata la struttura di massima dell'applicazione in termini di sottoinsiemi principali e loro interazione. Saranno tralasciate le fasi di sviluppo e la descrizione dettagliata delle classi.

4.1 Identificazione di oggetti e servizi

La composizione dei vari sottoinsiemi funzionali discende direttamente dalla struttura dell'algoritmo; la figura 19 riassume le principali classi.

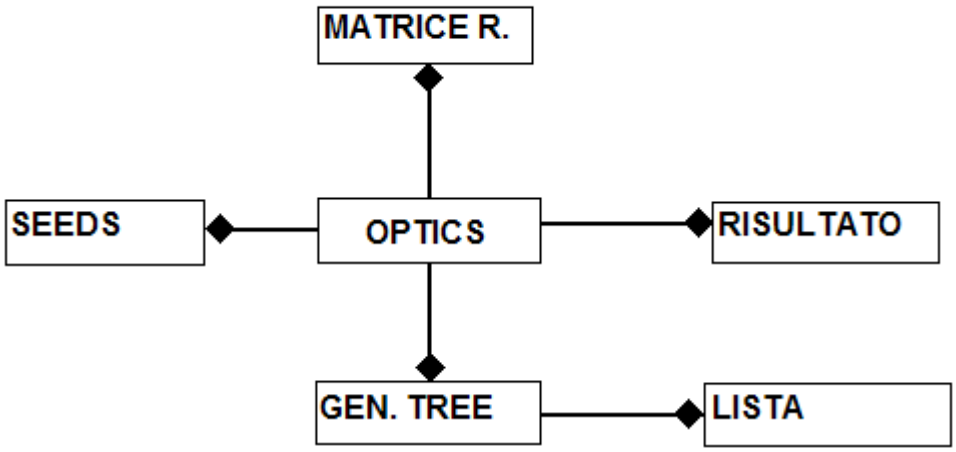

figura 19: Oggetti e servizi dell'applicazione

La classe **`matrice R.`** contiene le utilità per memorizzare la matrice di relazione e per accedere ai vari elementi. A partire da una struttura quadrata, occorre memorizzare i vettori, con lunghezza in ordine crescente rappresentanti la matrice triangolare inferiore

corrispondente alla matrice delle somiglianze. Nel caso in cui la matrice non sia perfettamente simmetrica, ogni valore memorizzato rappresenta la media dei corrispondenti nella matrice di ingresso. Si osserva sperimentalmente che utilizzare il valore medio piuttosto che il massimo (o il minimo) comporta risultati migliori[14]. Il nucleo dell'elaborazione è costituito dalla classe **`optics`** che produce il vettore delle reachability-distance e trova i vari cluster annidati; ogni cluster trovato è inserito nell'albero generico rappresentato dalla classe **`gen.Tree`**. Quest'ultima si occupa di trovare la lista di cluster non sovrapposti[15] e di assegnare ad ogni pattern l'appartenenza al cluster. La classe Lista è utile per ordinare le foglie dell'albero per qualità e per segnare le migliori che saranno espanse fino a contenere tutti i punti. La classe **`seeds`** si occupa di mantenere in ordine di distanza i punti del vicinato da elaborare. La classe r**`isultato`** permette di gestire il vettore delle reachability-distance generato da **`optics`**. Per quanto riguarda i parametri in ingresso l'applicazione deve consentire all'utente di impostare i valori di *minPts*, ε, ξ, ν. Tra gli aspetti che non vengono gestiti dalle classi citate nella figura precedente si ricorda la procedura che, a partire dal vettore della corretta classificazione, riporta il numero di errori commessi da ROPTICS; tale classe risolve anche il problema della corrispondenza tra cluster ed etichette. In pratica si crea una matrice di classificazione $C \times C$ in cui ogni (i,j) è incrementato, scorrendo i vettori delle appartenenze dei pattern ai cluster, ogni volta che l'elemento del cluster i viene assegnato da ROPTICS al cluster con etichetta j; al termine la corretta corrispondenza viene trovata in base ai massimi valori della matrice calcolati per ogni riga. Trovata tale corrispondenza 'ottimale' le classificazioni che non la rispettano sono considerate errori di classificazione e quindi la somma di questi casi produce il numeri di errori totale. Dato che la struttura dell'applicazione è facilmente scomponibile in moduli separati si è ritenuto opportuno ricorrere ad un linguaggio orientato agli oggetti che mantenga una certa efficienza nell'uso delle risorse hardware:il C++.

14 Ad esempio nel caso dell'insieme di dati relational cc con matrice prodotta dal sistema di estrazione delle somiglianze (vedere paragrafo 5.2): scegliendo il massimo o gli elementi di uno dei due triangoli non si riesce a distinguere bene i cluster.

15 Si noti che, come è stato detto, i cluster sono espressi tramite il punto di inizio e fine rispetto all'ordine prodotto da OPTICS, quindi occorre tenere traccia della corrispondenza tra vecchio e nuovo identificatore del punto.

Capitolo 5

Risultati sperimentali

Al fine di valutare la bontà del nuovo algoritmo occorre realizzare una serie di prove su vari dati. Tra le altre cose verrà valutata l'influenza dei parametri in ingresso sul risultato. Nell'ambito della letteratura si fa spesso ricorso a insieme di dati noti e facilmente reperibili[16] nella letteratura scientifica, in modo che gli esperimenti siano riproducibili. Nel seguito si farà riferimento anche a dati sintetici prodotti autonomamente.

5.1 Test su dati sintetici

Al fine di valutare l'incidenza dei parametri viene riportato un esempio basato su un insieme di dati costruito artificialmente. Si consideri, quindi, l'insieme di dati organizzati in modo non compatto e separato, come nella figura seguente.

```
Caratteristiche dell'insieme di dati:
numero pattern: 1000
numero cluster: 5
```

16 Si veda ad esempio il sito http://www.ics.uci.edu/~mlearn/MLSummary.html

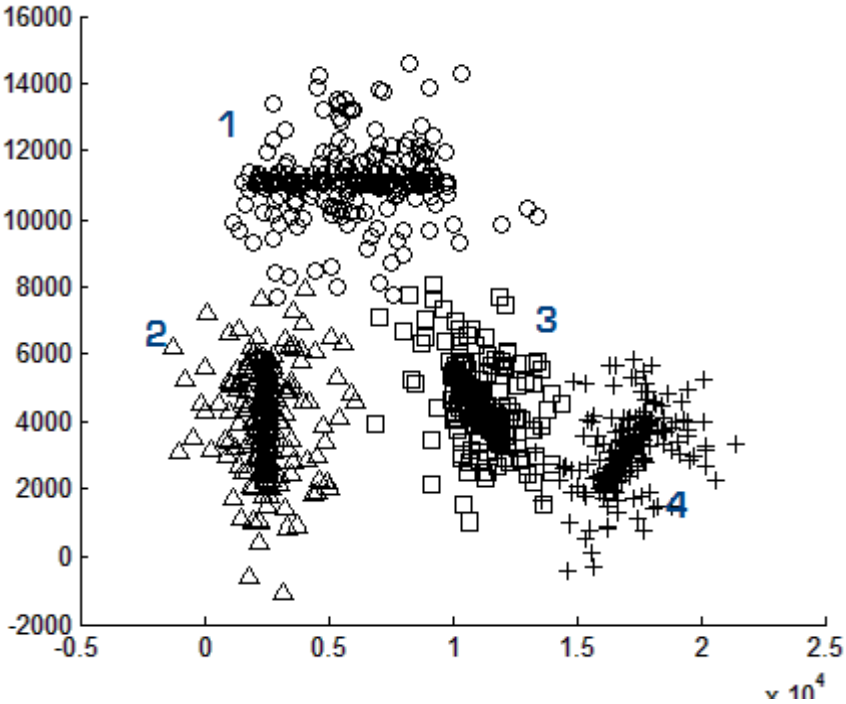

figura 20: Insieme di 1000 dati molto sparsi

L'esecuzione di ROPTICS, generando la matrice di relazione tramite la distanza euclidea, può essere riassunta con i dati seguenti:

```
minPts: 20
ξ: 0.0001
tasso di classificazione: non si riesce a separare i cluster
```

La mancata separazione di un cluster da un altro accade perché non si riesce ad accorpare in maniera corretta i piccoli cluster in modo da ricostruire quelli grandi come si vede in figura 21. La soluzione possibile consiste nell'aumentare il *minPts* in modo da limitare il numero di cluster (foglie) presenti nell'albero.

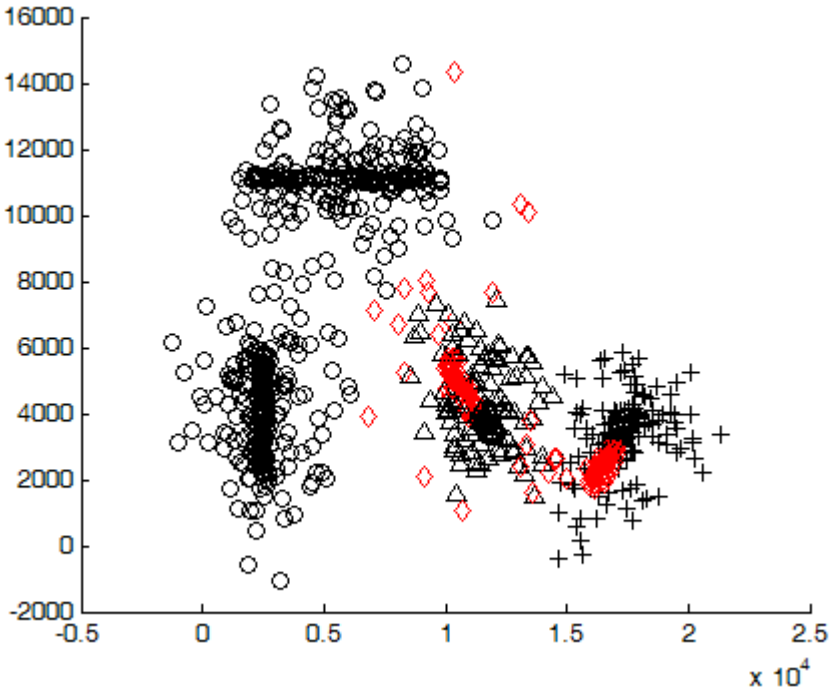

figura 21: Errori di ROPTICS dovuti a parametri sbagliati; i cluster 1 e 2 vengono considerati come cluster unico

Si consideri la seguente esecuzione di ROPTICS:

```
minPts: 80
ξ: 0.0001
tasso di classificazione: 98%
```

La figura 22 riporta, disegnati a forma di rombo, i pattern assegnati al cluster sbagliato.

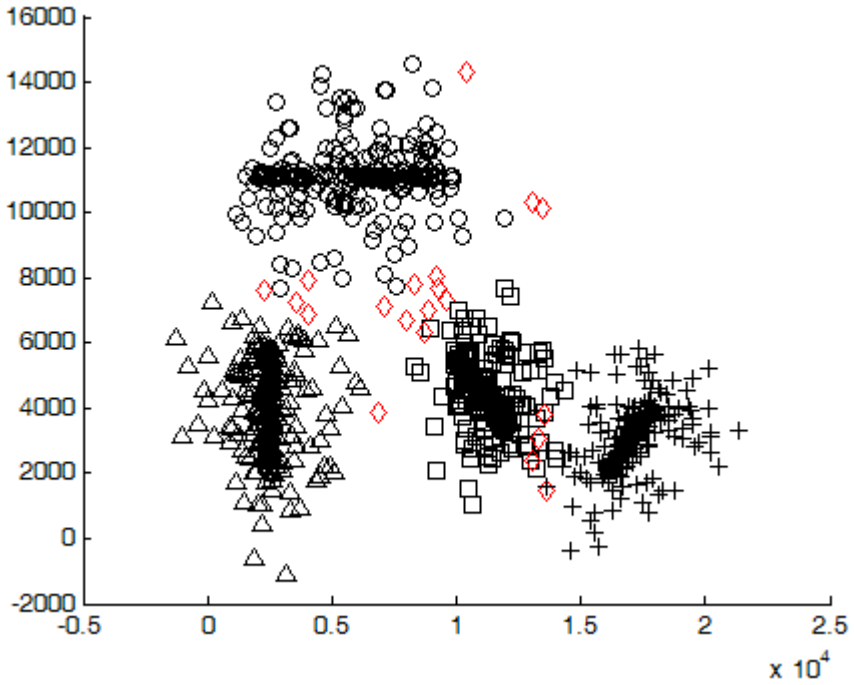

figura 22: Errori di ROPTICS con parametri corretti

Come era lecito aspettarsi i pattern erroneamente classificati sono quelli periferici.

Si noti che per evitare il problema si è sviluppata una versione alternativa della procedura di estrazione di *C* cluster dall'albero. In tale versione si prendono in considerazione tutti i nodi dell'albero ai fini dell'espansione; si segnano cioè non le foglie migliori ma i migliori elementi dell'insieme composto da nodi e foglie; si procede, infine, all'accorpamento dei pattern non presenti negli elementi segnati. Si nota che in questo ed in altri insiemi di dati (come IRIS) non ci sono differenze nel tasso di classificazione e risulta complicata la scelta di una misura adeguata di qualità dei cluster ottenuti. Per questi motivi si è deciso di tornare alla versione precedente che presenta maggiore efficienza.

5.2 Test su dati reali

Sono riportati dei test effettuati su dati provenienti dal mondo reale al fine di dimostrare il corretto funzionamento dell'algoritmo anche su dati non costruiti ad hoc.

Test sul data set Winsconsin Breast Cancer

La matrice relazionale utilizzata è stata estratta con un procedimento che, a partire da un sottoinsieme minimo di somiglianze, esegue una generalizzazione attraverso la logica

fuzzy e gli algoritmi genetici. Il data set originario contiene circa 700 pattern in dieci dimensioni rappresentanti parametri medici di analisi di tessuti tumorali Si vuole effettuare un clustering che, a partire dalle somiglianze tra tali analisi, riesca a categorizzare i pazienti sopravvissuti da quelli dei non sopravvissuti.

```
Le caratteristiche dell'insieme di dati:
Numero pattern: 683
Numero cluster: 2
```

Tramite Matlab™ si può ottenere una rappresentazione del tipo seguente (i punti di colore scuro indicano coppie di pattern simili):

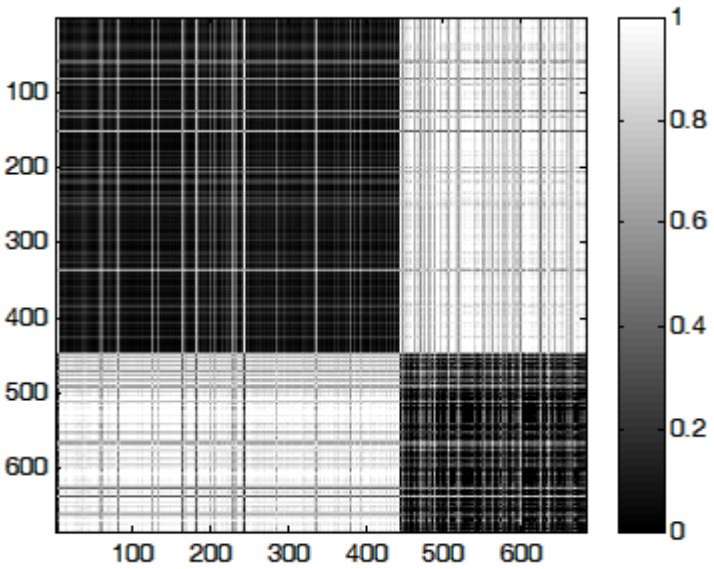

figura 23: Rappresentazione della matrice di dissimilarità

I risultati di ROPTICS con i relativi parametri sono riportati di seguito:

```
minPts: 20
ξ: 0.0001
Tasso di classificazione: 96.6%
```

Utilizzando un *minPts* diverso il numero di errori non cambia.

Test sul data set Relational cc

Si supponga di avere dei dati nello spazio $\mathbb{R}^2$ organizzati in due cluster e distribuiti come in figura 24.

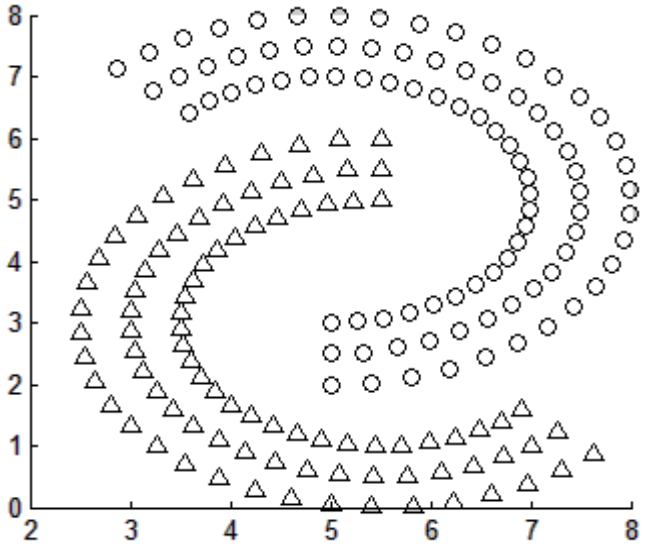

figura 24: Rappresentazione nello spazio oggetto del data set relational cc.

Si nota subito che in corrispondenza delle parti terminali di ciascun cluster a forma di "C", vi sono pattern topologicamente vicini a pattern della classe opposta, e molto lontani da pattern della medesima classe, situati all'altra estremità. Nella fattispecie, un algoritmo come OPTICS applicato nello spazio oggetto presenta prestazioni perfette, mentre è chiaro che algoritmi come FCM non possono classificare correttamente i pattern.

La matrice generata dall'algoritmo genetico con training set al 10% è rappresentata in figura 25.

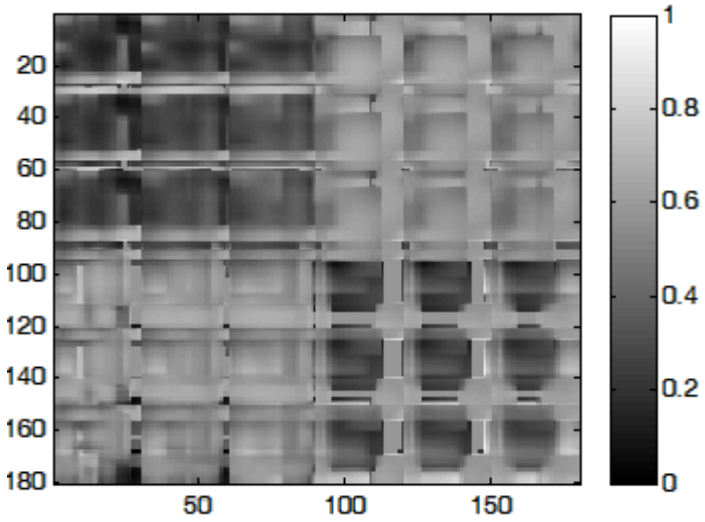

figura 25: Rappresentazione della matrice di relazione prodotta da un sistema di estrazione delle somiglianze

I risultati di ROPTICS con i relativi parametri sono:

```
minPts: 20
ξ: 0.0001
Tasso di classificazione: 90.55%
```

Utilizzando, invece, una matrice di relazione basata su distanza (di Manhattan) si ha una rappresentazione come la seguente:

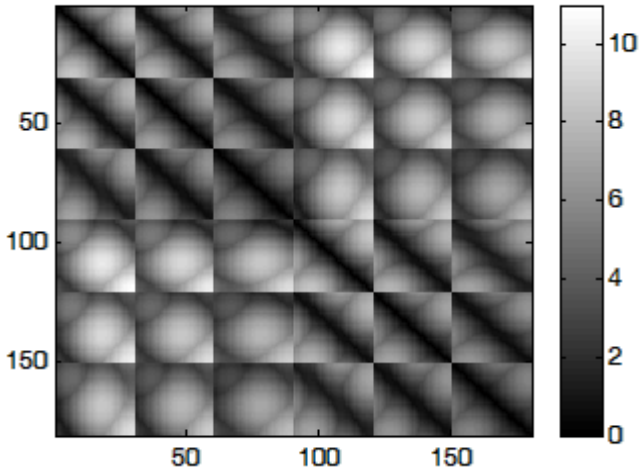

figura 26: Rappresentazione della matrice di relazione ottenuta con la distanza di Manhattan

Si noti come nei due casi la matrice sembri partizionata in sei caselle. Tale forma deriva dal fatto che i due cluster sono composti da tre strisce di punti e ciascuna striscia si compone di punti in ordine corrispondente ad una scansione in senso orario dei pattern in

figura. Quindi al termine di una striscia si passa al primo elemento della striscia successiva che si trova topologicamente dalla parte opposta.
I risultati di ROPTICS con i relativi parametri sono:

minPts: 20
ξ: 0.0001
Tasso di classificazione: 100%

Si vedrà in seguito che un algoritmo partizionale, che adopera una distanza (di Manhattan nella fattispecie), non produce buoni risultati.

5.3 Conclusioni

Come si è visto, in generale, l'algoritmo risulta sensibile al parametro *minPts*. In ogni caso, però, si nota come tale parametro risulti determinante ai fini della separazione o meno dei vari cluster, cioè se viene modificato di poche unità il tasso di classificazione rimane identico. In genere, se per il valore tipico di 20 ROPTICS non trova abbastanza cluster, allora occorre diminuire tale valore; se il numero di cluster trovati è corretto ma ROPTICS separa un certo cluster in diverse parti, può rendersi necessario aumentare questo parametro. Il parametro ξ, invece, non influenza in alcun modo i risultati a meno che non sia troppo alto (maggiore o uguale a 0.1).

Capitolo 6

Descrizione di algoritmi analoghi

Di seguito viene presentata una panoramica sulle alternative a ROPTICS presenti in letteratura. Come si vedrà, nella scelta di un algoritmo non si può prescindere dal dominio applicativo e non ci si può esimere da considerazioni relative alla complessità, la quale, nei problemi reali, gioca un ruolo fondamentale. Verranno descritti in dettaglio i metodi di riferimento cioè quelli su cui si baserà il confronto più rigoroso con ROPTICS. Questi algoritmi sono l'FCM ed il NE-FRC. Per gli altri algoritmi si riporta solo una presentazione generale, dato che in letteratura è già presente un confronto di tali algoritmi con FCM ed NE-FRC.

6.1 FCM - Fuzzy C-Means

Uno degli algoritmi più conosciuti in letteratura è il *C*-means (noto anche come *N*-means o *k*-means), le cui due varianti principali sono l'hard *C*-means ed il fuzzy[17] *C*-means. Il secondo rappresenta una generalizzazione del primo e sarà descritto di seguito.

Sia dato un insieme di oggetti $X=\{X_1,...X_N\}$ si definisce partizione fuzzy una famiglia di insiemi fuzzy $P=\{A_1,...,A_C\}$ tali che la somma delle appartenenze ai vari cluster vale sempre 1. Si supponga di rappresentare la partizione fuzzy con una matrice $U=[u_{ik}]$ con

$$u_{ik} \in [0,1] \quad \text{e} \qquad \sum_{i=1}^{C} u_{ik} = 1 \quad , \qquad 0 < \sum_{k=1}^{N} u_{ik} < N$$

17 La logica *fuzzy* o *logica sfumata* rappresenta una generalizzazione della logica booleana cioè i valori di verità di una proposizione vengono ad assumere un valore tra zero e uno. Nel clustering fuzzy un punto può avere gradi di appartenenza relativi a più cluster con il vincolo che la somma di tali valori è 1.

I vari cluster sono rappresentati da punti chiamati centroidi, i centroidi vengono individuati da FCM in modo da minimizzare una certa funzione obiettivo; in tal senso l'FCM si presta bene alla ricerca di cluster circolari (o ipersferoidali).
Sia dato il coefficiente $m \in [1, \infty)$ detto *costante di fuzzificazione* o *fuzziness;* l'algoritmo ha la struttura che segue:

1. Generare una partizione P casuale
2. Calcolare le coordinate dei centroidi tramite la formula seguente:

$$V_{ij} = \frac{\sum_{k=1}^{N} u_{ik}^{m} X_{kj}}{u_{ij}^{m}}$$

   in cui V_{ij} è la coordinata j-esima del vettore rappresentante il centroide del cluster i-esimo.
3. Aggiornare la partizione tramite la formula seguente:

$$u_{ik} = \frac{1}{\sum_{j=1}^{C} \frac{\left\| X_k - V_i \right\|^2}{\left\| X_k - V_j \right\|^2}}$$

   con i=1,...,C e k=1,...,N
4. Iterare i passi 2 e 3 finché la distanza tra le due partizioni risulta inferiore ad una certa soglia. Per distanza si intende la quantità[18] $d(P^i, P^j) = \left\| U^i - U^j \right\|$. Il processo, quindi, ha termine quando $d(P^m, P^{m-1}) < \varepsilon$ con ε errore massimo ammissibile.

Le formule di cui al punto 2 e 3 rappresentano una procedura di ottimizzazione alternata della funzione obiettivo seguente:

$$J_m(U, V) = \sum_{i=1}^{C} \sum_{k=1}^{N} u_{ik}^{m} \left\| X_k - V_i \right\|^2$$

Ad ogni iterazione vengono minimizzate le espressioni riportate di seguito:

18 La norma di una matrice può essere definita in vari modi. Ad esempio la norma della matrice A=[a_{ij}] si indica con $\|A\|$ e può essere definita nel seguente modo: $\|A\| = \max_j \sum_{i=1}^{n} |a_{ij}|$ (questa definizione è detta *norma 1*)

- $J_m(\cdot, V)$ con V fissato corrisponde alla formula al punto 3
- $J_m(U, \cdot)$ con U fissato corrisponde alla formula al punto 2

Per *m*=1 si ha l'FCM hard in cui ogni elementi appartiene ad uno solo dei cluster con grado 1. Poiché non esistono formule per calcolare il valore ottimale di *m*, occorre effettuare diverse prove per diversi valori di questo parametro. Di solito, nella pratica, non si conosce nemmeno il numero di cluster per cui vengono effettuate varie prove anche per la ricerca di tale parametro.
La complessità di FCM è *O*(*NCL*) in cui *C* è il numero di cluster e *L* è il numero massimo di iterazioni fissato dall'utente. Si può concludere che la complessità aumenta linearmente all'aumentare dei dati.

6.2 RFCM – Relational FCM

Per quanto concerne la versione relazionale di FCM, solitamente si ricorre ad una trasformazione dell'espressione al paragrafo precedente in una funzione K_m chiamata duale relazionale di J_m.:

$$K_m(U) = \sum_{i=1}^{C} \left(\sum_{j=1}^{N} \sum_{k=1}^{N} \left(u_{ij}^m u_{ik}^m \| x_j - x_k \|^2 \right) / \left(\sum_{t=1}^{N} 2u_{it}^m \right) \right)$$

L'espressione $r_{ij} = \| x_j - x_k \|^2$ coincide coi vari valori di dissimilarità della matrice di partenza; si noti come tali valori di dissimilarità debbano essere necessariamente delle misure di distanza euclidea.

6.3 FCMdd – Fuzzy C-medoids

La caratteristica più significativa dell'FCMdd consiste nella sua maggiore efficienza nei confronti dell'FCM. FCMdd non si basa su centroidi ma su *medoid*, cioè i centroidi sono costituiti dal pattern che meglio rappresenta il cluster. L'insieme dei medoid è indicato nel seguito con $V = \{v_1, ..., v_c\}$. *V* è un sottoinsieme del data set *X* con cardinalità *C* e X^c è l'insieme di tutti i sottoinsiemi *V* possibili. La funzione obiettivo che viene minimizzata è di questo tipo:

$$J_m(V, X) = \sum_{j=1}^{N} \sum_{i=1}^{C} u_{ij}^m r(x_j, v_i)$$

in cui *r* è la dissimilarità tra due oggetti ed *u* il grado di appartenenza fuzzy al cluster. La precedente formula non può essere minimizzata con procedura alternata; in genere sarà necessario provare tutti gli elementi di X^C e calcolare i gradi di appartenenza con una funzione del tipo seguente:

$$u_{ij} = \frac{\left(\frac{1}{r(x_j, v_i)}\right)^{1/(m-1)}}{\sum_{k=1}^{C} \left(\frac{1}{r(x_j, v_i)}\right)^{1/(m-1)}}$$

in cui *m* rappresenta la fuzzyness.

L'algoritmo svolge ciclicamente i passi seguenti:

- Calcola le appartenenze u_{ij}
- Calcola i nuovi medoid tramite la formula $v_i = x_q$ tale che:

$$q = \underset{1 \leq k \leq N}{argmin} \sum_{j=1}^{N} u_{ij}^m r(x_k, x_j)$$

L'algoritmo prosegue fino al numero di iterazioni fissato o finché il nuovo insieme dei medoid *V* è uguale al precedente.

Gli autori stimano la complessità di FCMdd in $O(N^2)$; per diminuire questo valore si fa in modo da aggiornare i medoid in base ad un sottoinsieme dei migliori *p* pattern assegnati al cluster del medoid preso in esame. Con questo accorgimento, la complessità risulta $O(NCp)$.

6.4 R-NE-FRC - Robust Non-Euclidean Fuzzy Relational data Clustering

La prima caratteristica di R-NE-FRC, consiste nel fatto che i valori della matrice di dissimilarità non devono necessariamente essere dei valori di distanza euclidea; inoltre si ha robustezza nei confronti del rumore. Di seguito viene riportata la struttura dell'algoritmo con *C* numero di cluster e *N* punti. Occorre, anzitutto, selezionare un valore del parametro δ; si suppone che il prototipo del rumore sia un prototipo “virtuale”

equidistante da tutti gli altri, e δ rappresenta questa distanza. In genere la ricerca di questo valore non è banale. Nei casi pratici si sceglie

$$\delta^2 = \lambda \left[\frac{\sum_{i=1}^{C} \sum_{k=1}^{N} d_{ik}^2}{CN} \right] \quad (1)$$

In cui λ rappresenta uno scalare il cui valore dipende dai dati presi in esame.
Per eseguire le operazioni si usano le seguenti formule:

$$a_{ik} = \frac{m \sum_{j=1}^{N} u_{ij}^m R_{jk}}{\sum_{j=1}^{N} u_{ij}^m} - \frac{m \sum_{h=1}^{N} \sum_{j=1}^{N} u_{ij}^m u_{ih}^m R_{jh}}{2 \left(\sum_{j=1}^{N} u_{ijv}^m \right)^2} \quad (2)$$

$$b_{ij} = a_{ij} u_{ij}^{m-2} \qquad (3)$$

La partizione dei cluster è calcolata come segue (I^+ e I^- sono due insiemi):

$$\begin{aligned} u_{ij} &= 0 \quad \text{per} \qquad i \in I^- \\ u_{ij} &= \frac{1/b_{ij}}{\sum_{w \in I^+} (1/b_{wj})} \quad \text{per } i \in I^+ \end{aligned} \qquad (4)$$

Per aggiornare i due insiemi I^+ e I^- si usa l'espressione seguente.

$$B_i = \frac{1/b_{ij}}{\sum_{w=1}^{C+1} (1/b_{wi})} \qquad (5)$$

L'aggiornamento avviene secondo le due regole:

- se $B_i \leq 0$ $\quad I^- = I^- \cup \{i\}$
- se $B_i > 0$ $\quad I^+ = I^+ \cup \{i\}$

Il termine seguente rappresenta la dissimilarità del rumore.

$$b_{*k} = \left[\frac{\sum_{i=1}^{C} \sum_{k=1}^{N} b_{ik}}{CN} \right]$$

L'algoritmo ha la struttura che segue:

```
Inizializzare i set di cluster I⁻ = I⁺ = Ø;
iter = 0;
Inizializzare una partizione fuzzy U=u_ik;
Ripeti
  for(1≤k≤n)
    for(1≤i≤C)
      Calcolare b_ik usando il nuovo u_jk per j<k ed il vecchio u_ij per j≥k;
      Calcolare B_i;
      Calcolare I⁻ ed I⁺ in base a B_i;
      Aggiornare la matrice u_ik per  i∈I⁺  e  i∈I⁻  ;
      Porre I⁻ = I⁺ = Ø;
    end
  end
iter = iter +1;
Finché ( ||U_old − U|| < ε  oppure iter = N_MAX_ITERAZIONI)
```

Al fine di calcolare la complessità asintotica di R-NE-FRC, si va a valutare la complessità del termine più costoso. Il numeratore della seconda frazione della formula 2 ha costo $O(M^2)$ ma è indipendente da k, quindi può essere precalcolato per ogni riga della matrice; il costo totale della formula (2) è $O(CM^2)$ con M numero di pattern e C classi.

6.5 NeRFCM – Non Euclidean Relational FCM

I limiti di RFCM, come si è visto, derivano dal fatto che può essere usato solo se i reciproci valori di dissimilarità della matrice R rappresentano delle distanze euclidee di N punti in un certo spazio. In particolare si dice che R è una relazione Euclidea se esiste un insieme di dati $X=\{x_1,\dots,x_N\}$ $X\subset\mathbb{R}^{n-1}$ tale che $R=\left[r_{ij}=\|x_j-x_k\|^2\right]$.
L'algoritmo NeRFCM non richiede che la relazione sia Euclidea ma solo che la matrice R sia a valori positivi con diagonale nulla e simmetrica. L'idea di base dell'algoritmo consiste nel trasformare R in una matrice Euclidea applicando, poi, RFCM.

6.6 ARCA – Any Relational Clustering Algorithm

ARCA rappresenta l'evoluzione dell'FCM per l'utilizzo con matrici di relazione. I centroidi sono punti le cui relazioni con gli altri punti sono rappresentative delle

differenze degli oggetti appartenenti a cluster diversi. ARCA, come del resto FCM, forma la partizione fuzzy dei cluster in modo da minimizzare la distanza Euclidea tra centroidi ed elementi del cluster.
La funzione che viene minimizzata ha la forma seguente:

$$J_m(U, V) = \sum_{i=1}^{C} \sum_{k=1}^{N} u_{ik}^m \delta^2(x_k, v_i)$$

In cui:

$$\delta(x_k, v_i) = \sqrt{\sum_{s=1}^{N} (r_{ks} - v_{is})^2}$$

dove $r_{k,s}$ è la dissimilarità tra i pattern x_k e x_s; v_i è il centroide del cluster i.
Come per FCM l'algoritmo segue una procedura di ottimizzazione alternata calcolando di volta in volta i nuovi prototipi e la nuova partizione con formule analoghe.

6.7 FNM – Fuzzy Non Metric Model

Questo algoritmo si utilizza per matrici di relazione simmetriche e si basa sulla funzione di ottimizzazione seguente:

$$K_{FNM}(U) = \sum_{i=1}^{C} \sum_{k=1}^{N} u_{ik}^2 D_{ik}$$

in cui:

$$D_{ik} = \sum_{j=1}^{N} u_{ij}^2 r_{kj}$$

L'algoritmo si basa sulla applicazione alternata dell'espressione precedente e della seguente:

$$u_{ik} = \left(\sum_{j=1}^{C} \frac{D_{ik}}{D_{jk}} \right)^{-1} \quad 1 \leqslant i \leqslant C;\ 1 \leqslant k \leqslant N$$

Lo scopo principale di FNM risiede nella possibilità di utilizzare matrici di relazione con caratteristiche non metriche. FNM in pratica può non funzionare in quanto

l'ottimizzazione si arresta, spesso, in un punto di minimo locale della funzione obiettivo.

6.8 AP – Assigment-Prototype

Supponendo di avere cluster di tipo crisp $X_1,...,X_c$, si assume che per ognuno di essi esista un punto o_{ki} dell'insieme di dati che meglio rappresenta il cluster. La qualità del clustering ottenuto è dato dalla formula che segue:

$$\tau = \sum_{i=1}^{C} \left(\sum_{o_i \in X_i} r_{jk_i} \right)$$

τ rappresenta una misura di dissimilarità tra gli oggetti dello stesso cluster nei confronti del prototipo; la minimizzazione della precedente formula porta ad un clustering di tipo hard. Per avere una partizione fuzzy si considera la formula riportata di seguito:

$$\underbrace{min}_{M_{fCN} \times M^*_{fCN}} \left\{ K_{AP}(U,T) = \sum_{i=1}^{C} \sum_{k=1}^{N} \sum_{j=1}^{N} u_{ik}^2 t_{ij}^2 r_{kj} \right\}$$

in cui

$$U \in M_{fCN}, M^*_{fCN} = \left\{ T \in \mathbb{R}^{C \times N} : \boldsymbol{T}_{(k)} \in N_{fN} \; \forall k \right\}$$

Nella precedente $\boldsymbol{T}_{(k)}$ rappresenta la k-esima riga della matrice T (matrice dei pesi dei prototipi) la quale rispetta la condizione $\sum_{k=1}^{N} t_{ik} = 1 \quad i = 1,..., C$ e $t_{ik} \geqslant 0 \; \forall i, k$.

U è la partizione fuzzy di O (insieme degli oggetti); e t_{ik} rappresenta il grado con cui o_k rappresenta il prototipo i-esimo. Come di consueto si usa la procedura di ottimizzazione alternata che deriva dall'applicazione del metodo dei moltiplicatori di Lagrange; le formule usate sono le seguenti.

$$t_{i\ell} = \frac{\left(1 / \sum_{k} u_{ik}^2 r_{k\ell} \right)}{\sum_{m} \left(1 / \sum_{k} u_{ik}^2 r_{km} \right)} \quad \forall i, \ell$$

$$u_{ik} = \frac{\left(1 / \sum_{\ell} t_{i\ell}^2 r_{k\ell} \right)}{\sum_{j} \left(1 / \sum_{\ell} t_{j\ell}^2 r_{k\ell} \right)} \quad \forall i, k$$

La procedura termina quando la differenza tra partizioni successive è minima.

Capitolo 7

Confronto tra i vari algoritmi

Confrontare diversi algoritmi significa effettuare varie prove su diversi data set e per diversi parametri utente; si deve tenere conto del fatto che alcuni algoritmi sono particolarmente sensibili ai parametri in ingresso; inoltre, due esecuzioni successive sullo stesso insieme possono produrre, anche con gli stessi parametri, risultati diversi (si pensi ad esempio all'FCM che inizia con una partizione casuale all'inizio). Per cui occorre eseguire diverse prove con i medesimi parametri e valutare i valori medi e le varianze dei risultati. In alcuni casi non si è persino sicuri che l'algoritmo termini in un numero finito di iterazioni. E quindi in tali casi si prende come partizione finale quella ottenuta dopo un certo numero prefissato di iterazioni. Occorre precisare che nella scelta di un algoritmo non si può prescindere dal dominio applicativo.

Nelle pagine che seguono si è scelto di confrontare i vari approcci in base a insiemi di dati noti in letteratura. I vari test su dati sintetici sono stati svolti a partire da punti in $\mathbb{R}^2$ generando le matrici relazionali tramite la distanza di Manhattan.

7.1 Confronto con FCM

Come è noto FCM risulta poco adatto per dati molto sparsi e agglomerati di punti dalla forma non convessa; inoltre, se la matrice di dissimilarità deriva da punti con un numero di features pari a qualche decina spesso non si è in grado di distinguere i cluster per i noti problemi legati all'alta dimensionalità. Quest'ultimo problema (dopo aver trasformato i dati in matrice relazionale) è presente anche per ROPTICS, mentre se si hanno cluster non convessi oppure con densità diverse un algoritmo basato sulla densità risulta migliore. Per FCM si intende l'FCM classico nello spazio oggetto; ad esempio se

abbiamo una matrice relazionale di 500 elementi FCM viene fatto girare su 500 pattern con 500 feature; si verifica che questa prassi produce risultati uguali a quelli della versione relazionale di FCM.

Quello che si osserva è che su dati regolari le prestazioni di FCM rimangono stabili a lungo ad esempio convertendo le coordinate sul piano di 11000 elementi (raffigurato in basso) in matrice relazionale si ha un tasso di classificazione del 100% ed un coefficiente di partizione del 99% per *m*=1.1.

```
Caratteristiche dell'insieme di dati e risultati:
numero pattern: 11000
cluster:3
criterio di arresto: 0.1
m: 1.1
Tasso di classificazione: 100%
Coefficiente di partizione: 99%
```

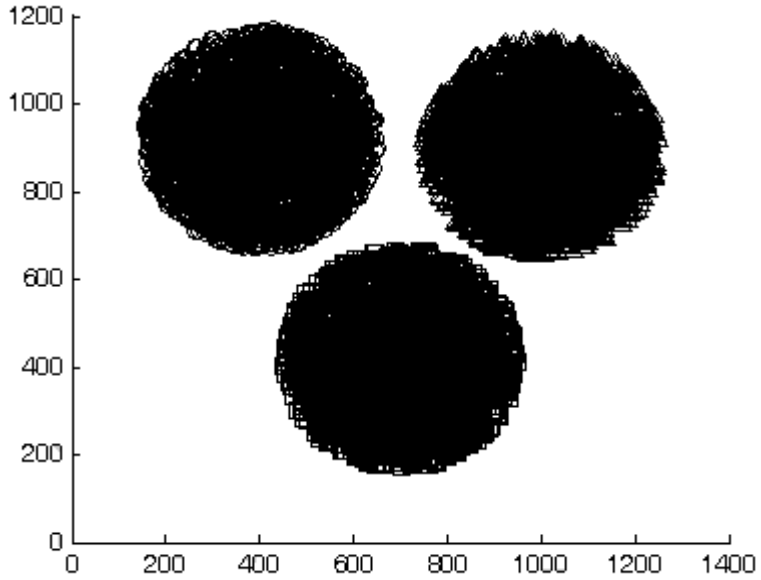

figura 27: Insieme costituito da 11000 pattern

Aumentando la fuzziness a 2 il tasso di classificazione si mantiene inalterato mentre il coefficiente di partizione scende a 0.788. Per *m*=3 si ha un coefficiente di partizione pari a 0.538.

Utilizzando l'insieme di dati di cui alla figura 20 si ottengono ottimi risultati anche aumentando i punti.

I limiti di FCM riguardano il suo utilizzo in situazioni in cui i cluster non siano distribuiti uniformemente nello spazio; si consideri allora un insieme di dati composto da cinque cluster di cui tre equidistanti e non sovrapposti a densità maggiore e due a densità minore:

```
Caratteristiche dell'insieme di dati:
numero pattern: 100
cluster:5
criterio di arresto: 0.0001
```

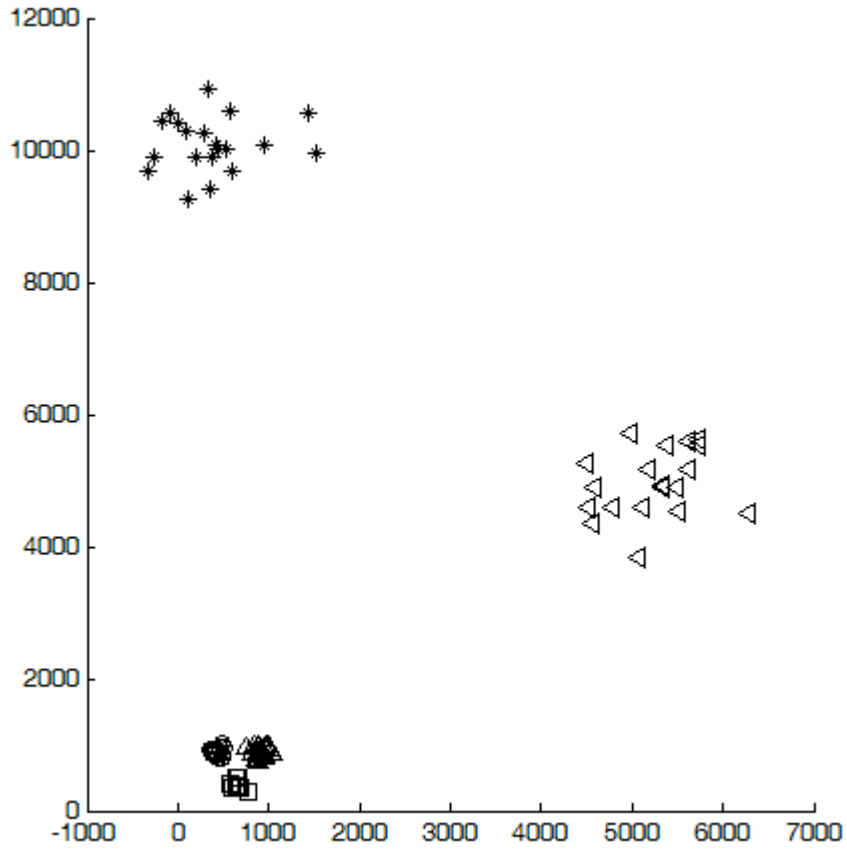

figura 28: Insieme di dati con cluster di diverse densità e non equidistanti

Eseguendo per m = [1.1,3.0] si ottiene un tasso di riconoscimento che presenta molte oscillazioni. I dati riportati nel grafico rappresentano i valori statistici dell'esecuzione di dieci prove per ogni valore di m. Le linee verticali quantificano la deviazione standard che si è riscontrata complessivamente nelle varie prove.

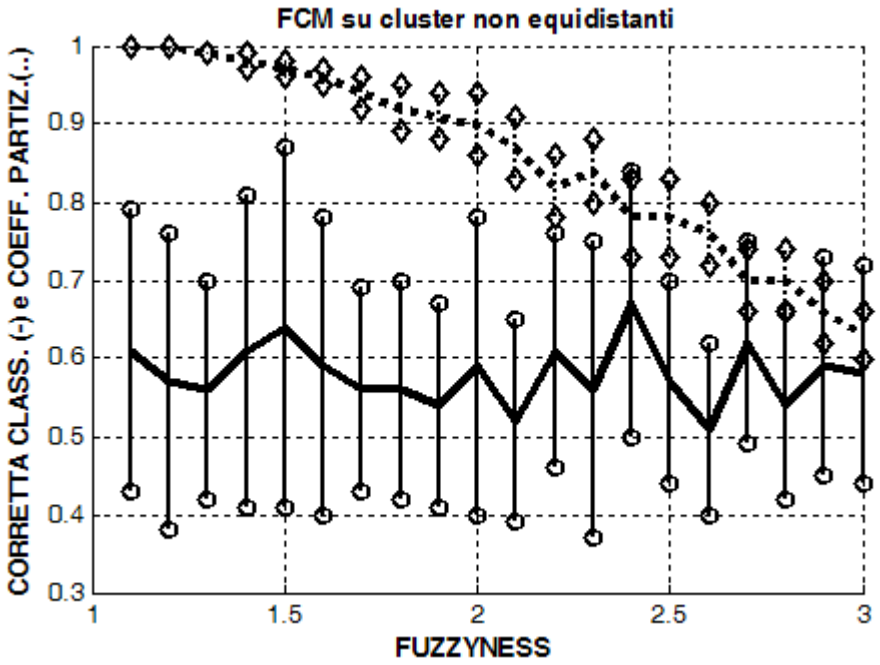

figura 29: Andamento del tasso di riconoscimento del clustering effettuato sull'insieme di dati della figura precedente

Durante le prove è capitato di ottenere la corretta classificazione in modo occasionale; se ne deduce che, in casi come questo, la buona riuscita dell'elaborazione è legata alla posizione iniziale dei centroidi (scelta casualmente all'inizio) cioè se i centroidi sono casualmente inizializzati in zone locali dense non riescono ad uscirne. Ad esempio se due centroidi sono posizionati all'interno di uno dei cluster poco densi tenderanno a rimanere in quella posizione. Quello presentato è il limite principale di FCM.

Per comprendere al meglio le sostanziali differenze tra ROPTICS de FCM, si consideri il data set relational cc proposto in precedenza (figura 24). Data la caratteristica fondamentale di FCM di trovare cluster convessi e tondeggianti è chiaro che applicare l'algoritmo nello spazio oggetto non porta ad una corretta classificazione (figura 30). Come si è detto ROPTICS identifica correttamente tutti i pattern se la matrice è costruita con distanza di Manhattan.

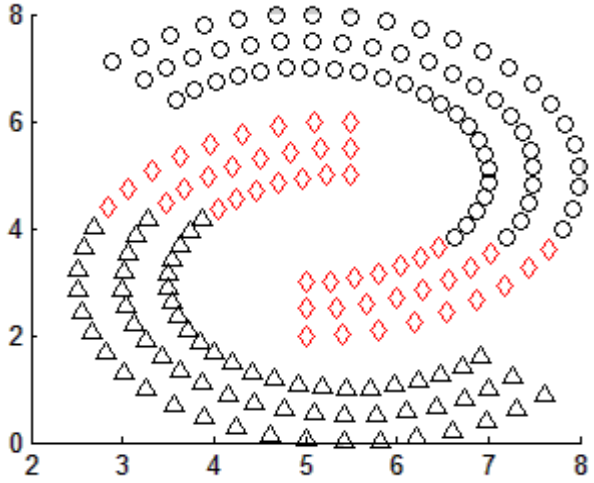

figura 30: Errori del clustering FCM nello spazio oggetto

Se si utilizza la matrice ottenuta tramite l'algoritmo genetico (matrice di relazione) si ha circa il 13% di errore (con *m* qualunque) contro il 13.3% di ROPTICS (*minPts*=20, ξ=0.0001).

Utilizzando il data set breast cancer si ottiene un tasso di classificazione costante al 96.778% di poco superiore a quello di ROPTICS (96.64% per *minPts*=20, ξ=0.001).

7.2 FCM ed estrazione delle somiglianze

Supponiamo di disporre del sistema di estrazione delle somiglianze dai dati basato su logica fuzzy ed ottimizzazione genetica, già citato in precedenza. Un parametro rilevante di tale applicazione riguarda la cosiddetta percentuale di training cioè al sistema viene fornito un corretto insieme di somiglianze già classificate in modo che, tramite un ciclo di ottimizzazione genetica possa generalizzare la somiglianza e rappresentarla mediante un insieme di regole fuzzy. Nelle pagine che seguono si mostreranno i risultati di FCM e ROPTICS al variare della percentuale di training. Per ogni percentuale esaminata vengono prese in esame due matrici, la prima è ottenuta dall'insieme di dati di training mediante una procedura di estrazione iniziale di regole fuzzy, la seconda è ottenuta dopo il ciclo di ottimizzazione genetica (indicato nel seguito anche con GA). Quando ROPTICS non è in grado di individuare i cluster significa che nel grafico delle reachability-distance non ci sono abbastanza avvallamenti.

Nelle pagine seguenti le prove con FCM sono state eseguite 5 volte per ognuno dei

valori da 1.1 a 3 della fuzziness.

Test effettuato con Iris

Per i test che seguono ROPTICS ha i seguenti parametri:

minPts: 5

ξ: 0.0001

Training set: 5% prima del GA

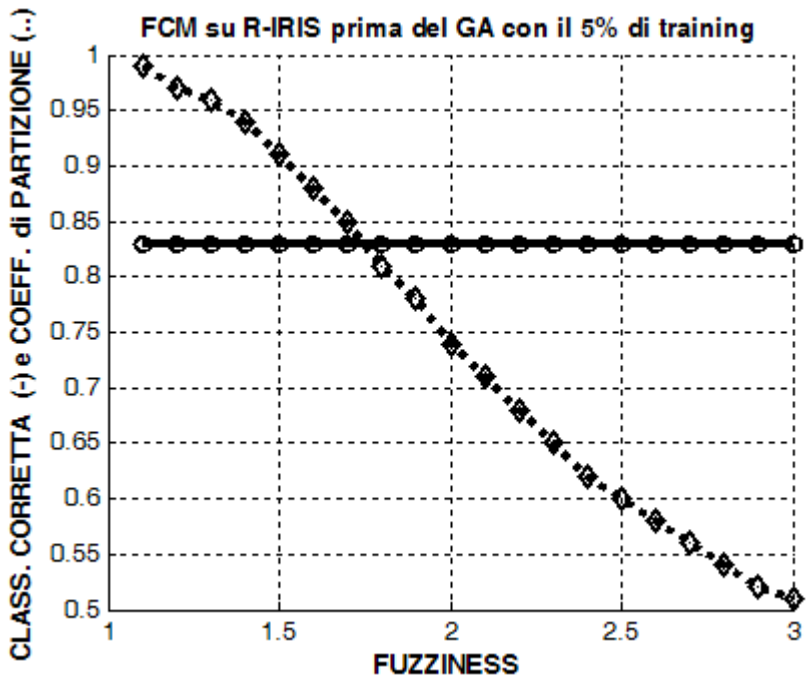

figura 31: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 5%

Risultati ROPTICS:

Impossibile trovare un numero sufficiente di cluster

Training set: 5% dopo il GA

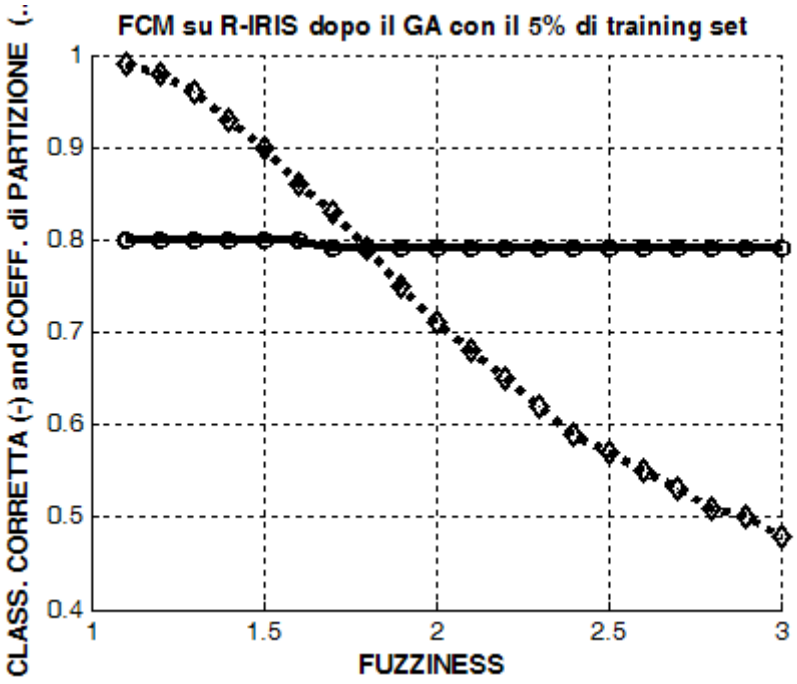

figura 32: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 5%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set 10% prima del GA

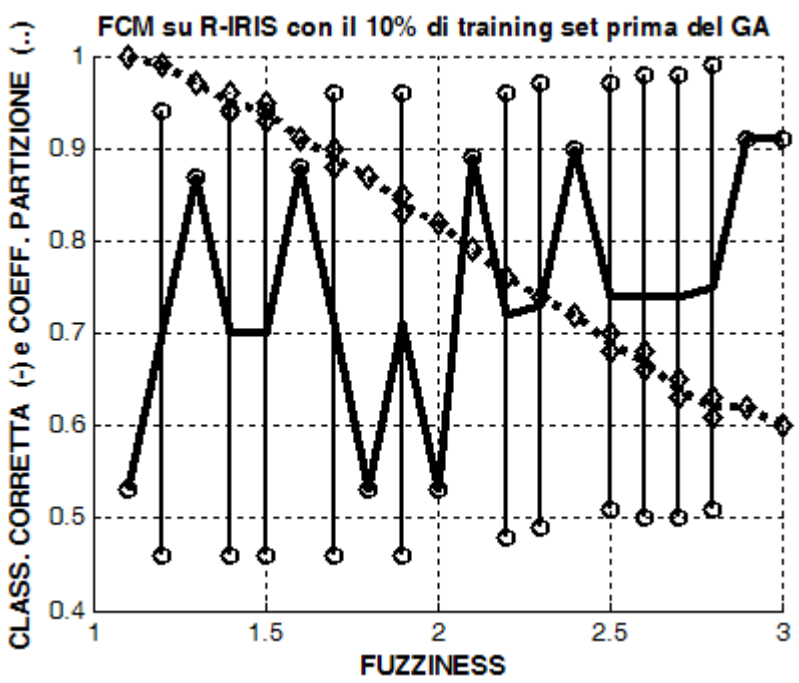

figura 33: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 10%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set 10% dopo il GA

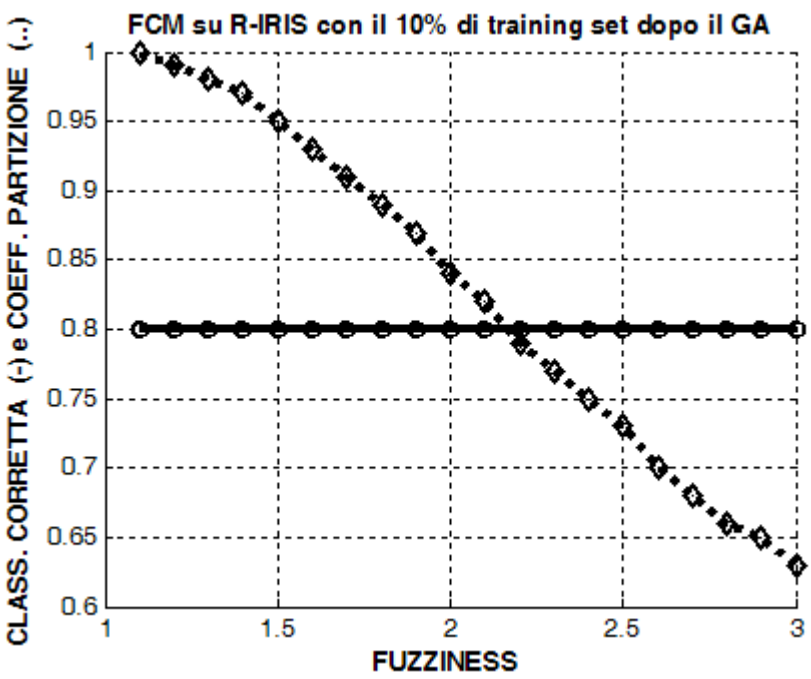

figura 34: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 10%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set 15% prima del GA

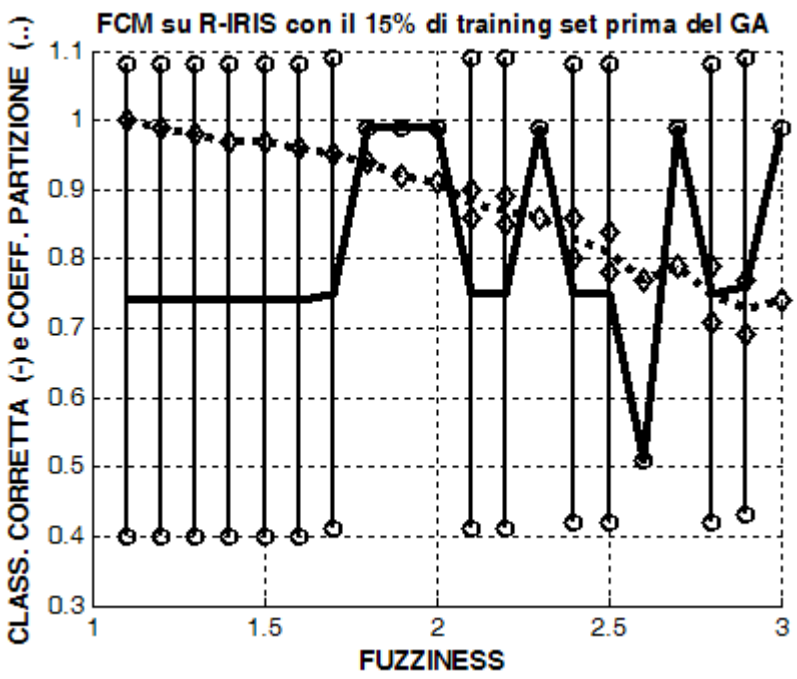

figura 35: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 15%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set 15% dopo il GA

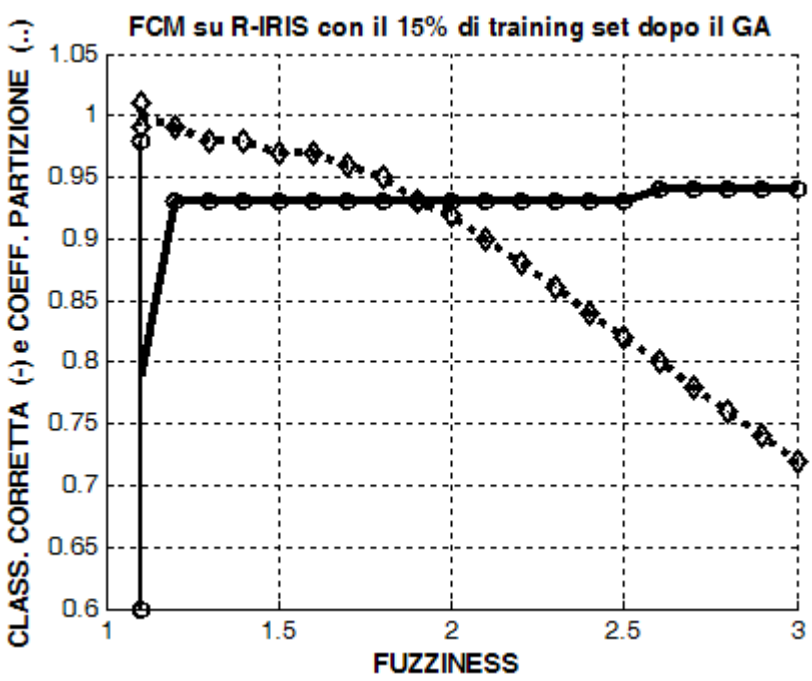

figura 36: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 15%

Risultati ROPTICS:

Tasso di classificazione 84%

Training set del 20% prima del GA

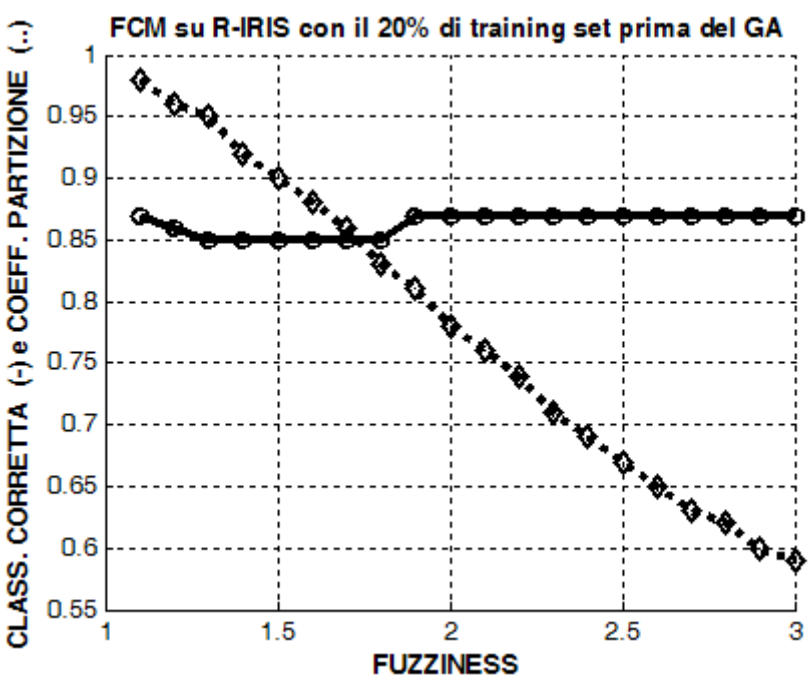

figura 37: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 20%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set del 20% dopo il GA

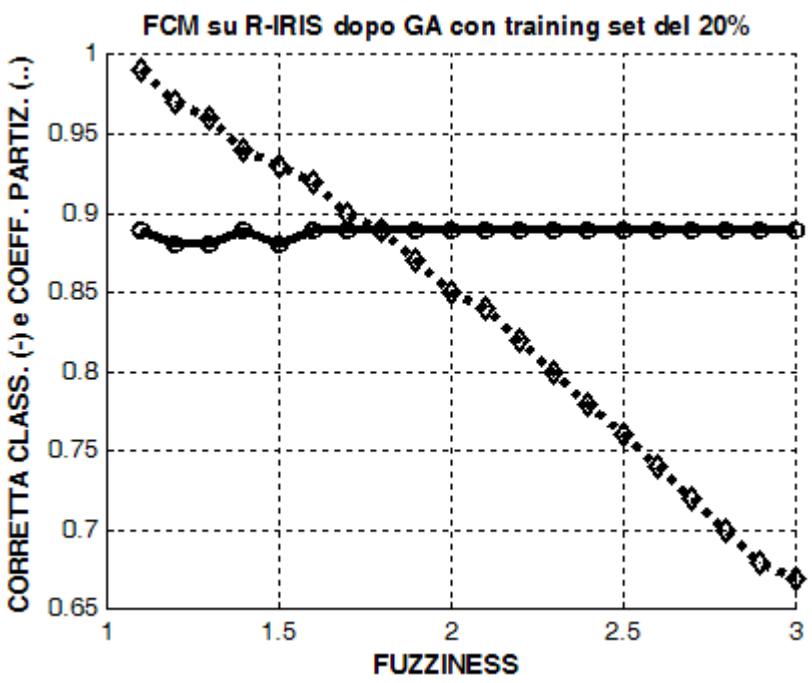

figura 38: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 20%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set del 25% prima del GA

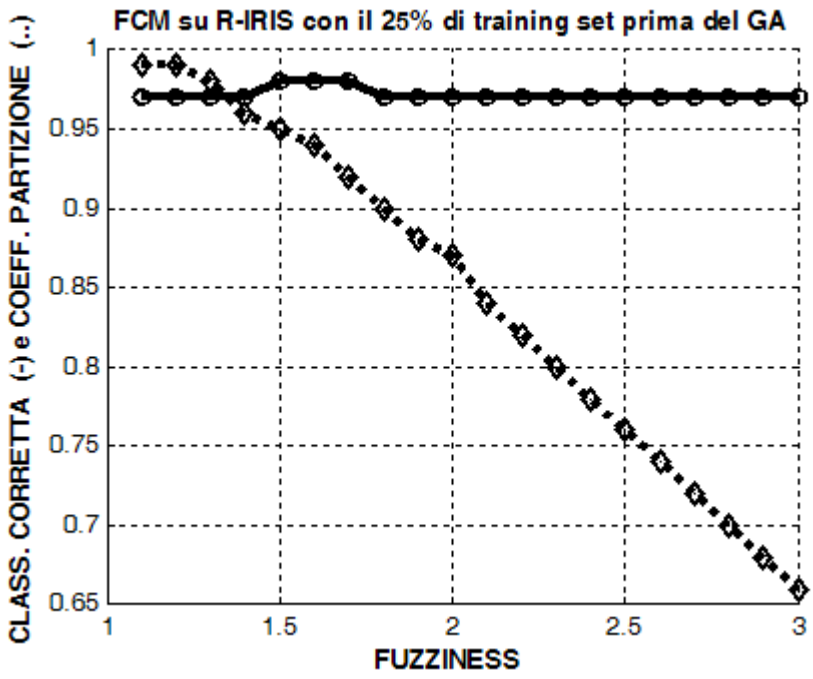

figura 39: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 25%

```
Risultati ROPTICS:
Impossibile trovare un numero sufficiente di cluster
```

Training set del 25% dopo il GA

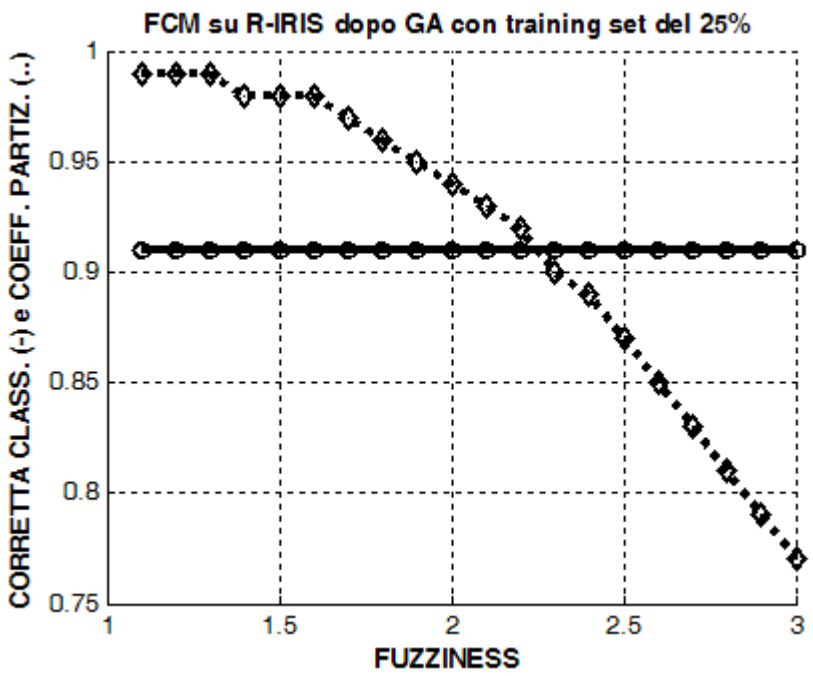

figura 40: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 25%

```
Risultati ROPTICS:
Tasso di classificazione: 94.6667%
```

Come si osserva dai risultati ROPTICS è migliore di FCM solo nell'ultimo caso. La tabella seguente riassume i risultati ottenuti.

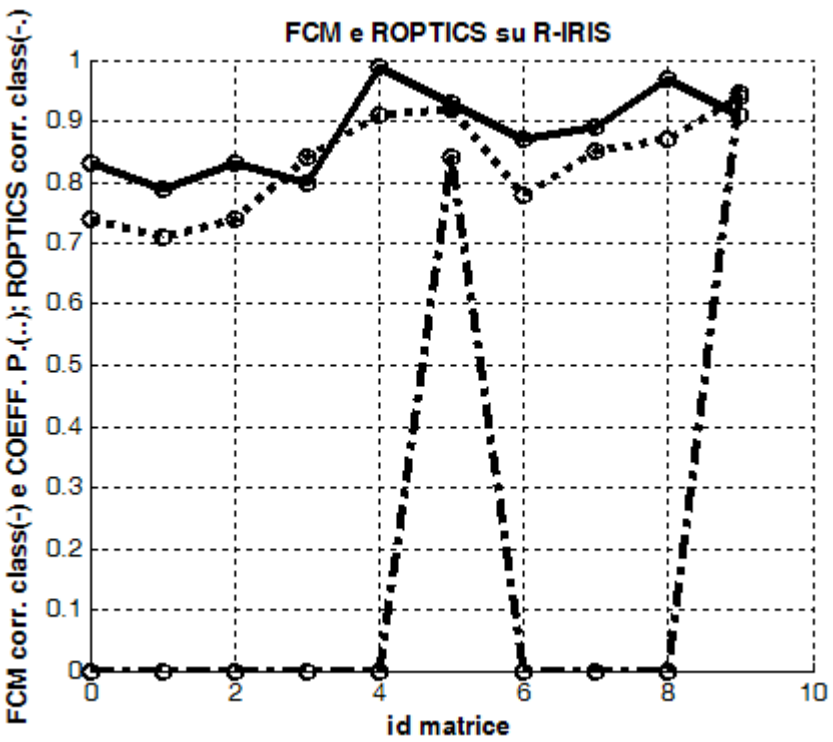

figura 41: Andamento del tasso di classificazione di FCM per *m*=2 (linea continua); del coefficiente di partizione di FCM (linea punteggiata) e del tasso di classificazione di ROPTICS (punto-linea). I valori sull'asse delle ascisse rappresentano il numero della matrice usata

L'identificatore della matrice (id matrice) risulta così associato:

0 -> training 5% prima del GA

1-> training 5% dopo il GA

2 -> training 10% prima del GA

3-> training 10% dopo il GA

4 -> training 15% prima del GA

5-> training 15% dopo il GA

6 -> training 20% prima del GA

7-> training 20% dopo il GA

8 -> training 25% prima del GA

9-> training 25% dopo il GA

Test effettuato su breast cancer

Per i test che seguono ROPTICS ha i seguenti parametri:

minPts: 20

ξ: 0.0001

Training set: 5% prima del GA

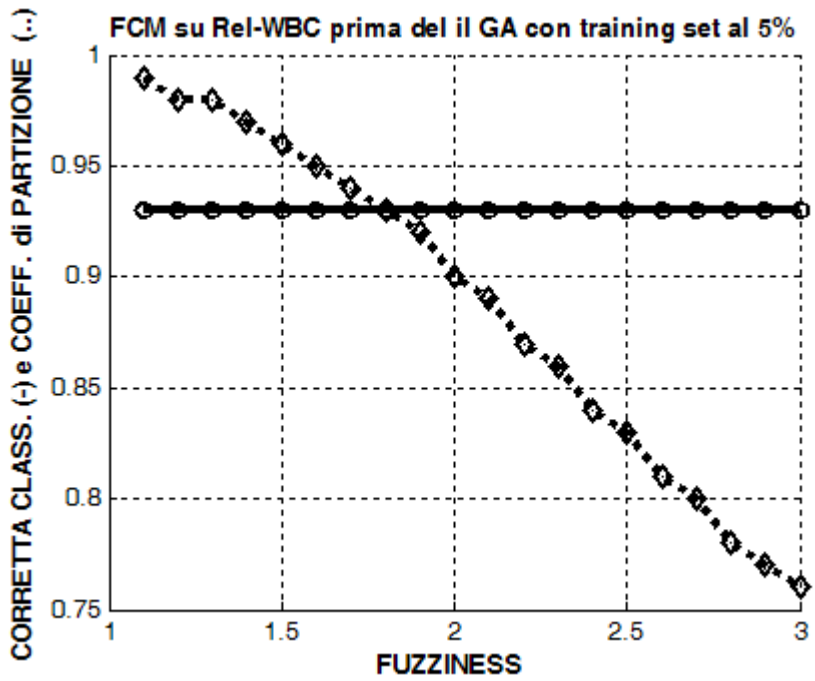

figura 42: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 5%

Risultati ROPTICS:

Impossibile trovare il numero specificato di cluster

Training set: 5% dopo il GA

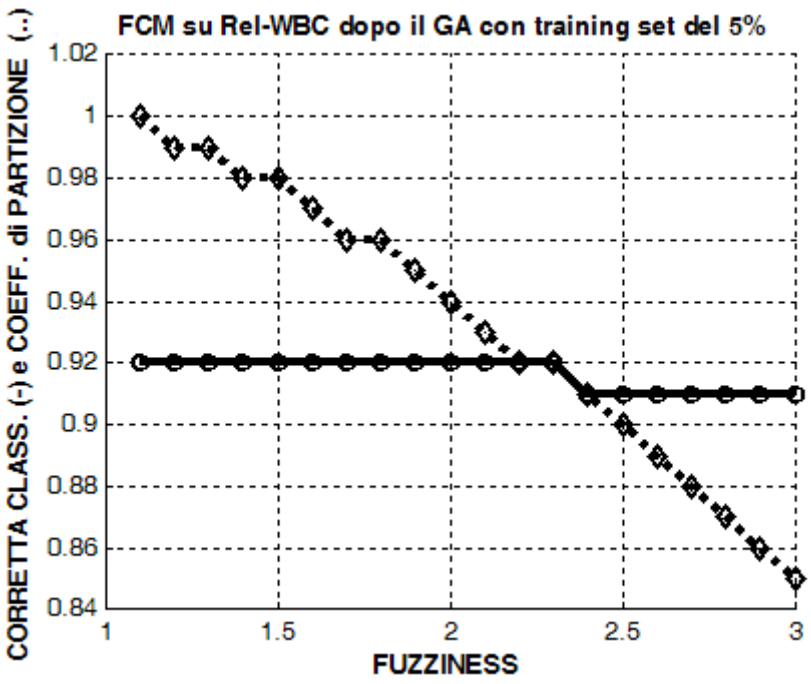

figura 43: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 5%

```
Risultati ROPTICS:
Tasso di classificazione: 93.7%
```

Training set: 10% prima del GA

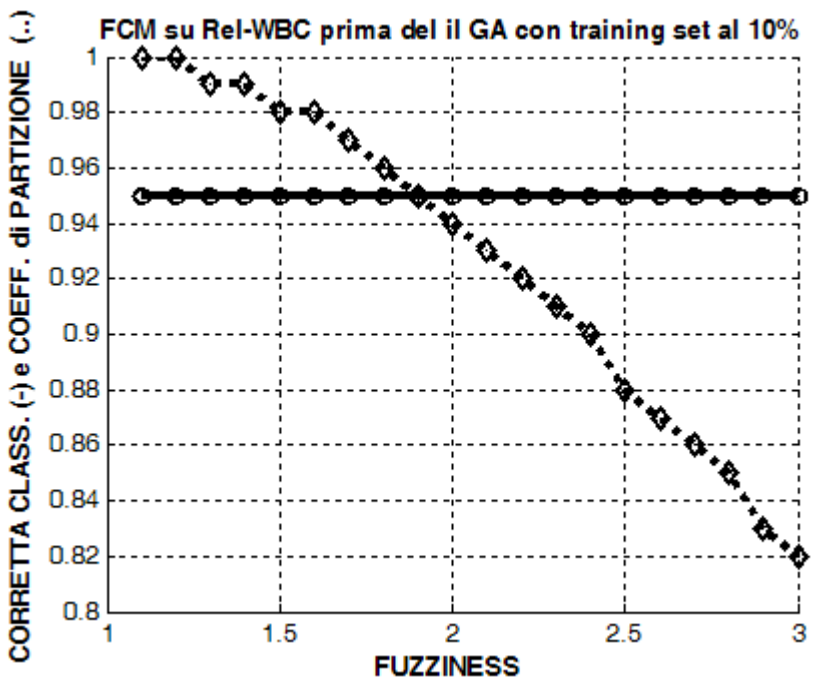

figura 44: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 10%

```
Risultati ROPTICS:
Tasso di classificazione: 95.022%
```

Training set: 10% dopo il GA

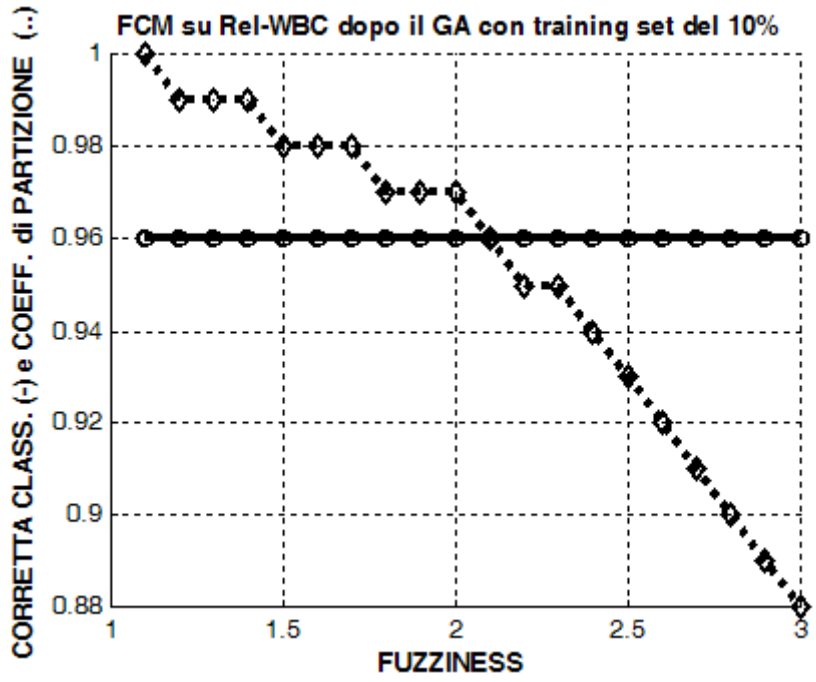

figura 45: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 10%

```
Risultati ROPTICS:
Tasso di classificazione: 95.16%
```

Training set: 15% prima del GA

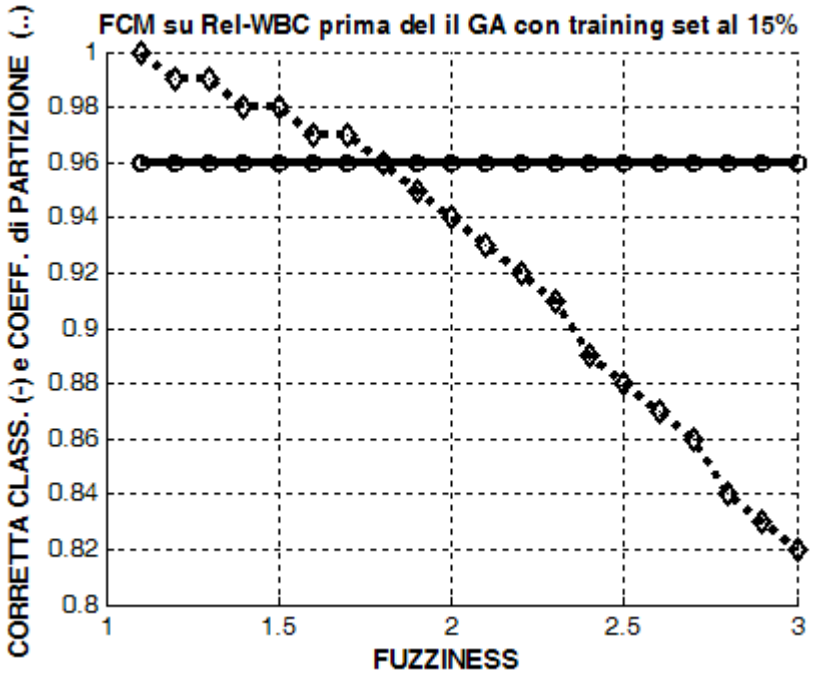

figura 46: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 15%

```
Risultati ROPTICS:
Tasso di classificazione: 93.7%
```

Training set: 15% dopo il GA

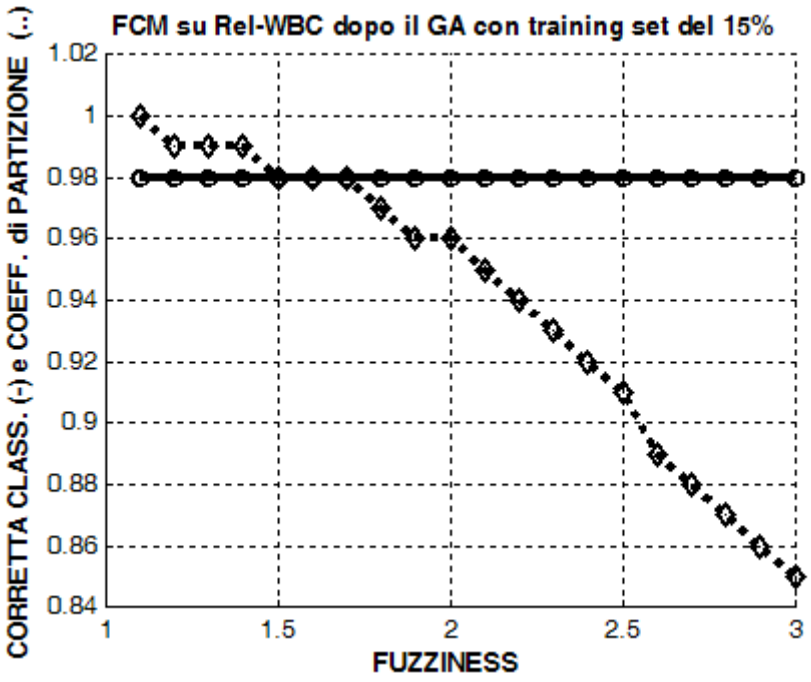

figura 47: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 15%

```
Risultati ROPTICS:
Tasso di classificazione: 96.19%
```

Training set: 20% prima del GA

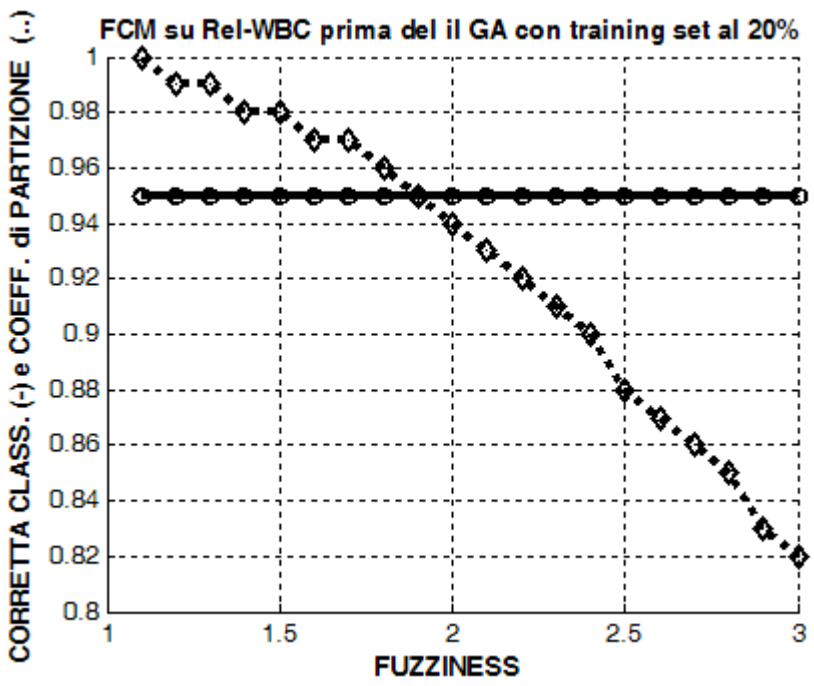

figura 48: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 20%

```
Risultati ROPTICS:
Impossibile trovare il numero di cluster specificato
```

Training set: 20% dopo il GA

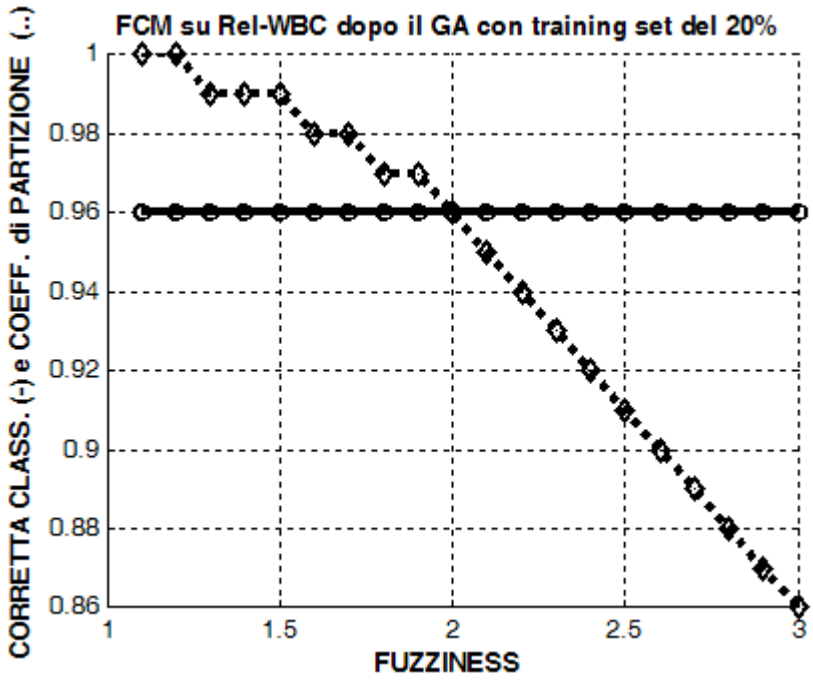

figura 49: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 20%

```
Risultati ROPTICS:
Tasso di classificazione: 97.2%
```

Training set: 25% prima del GA

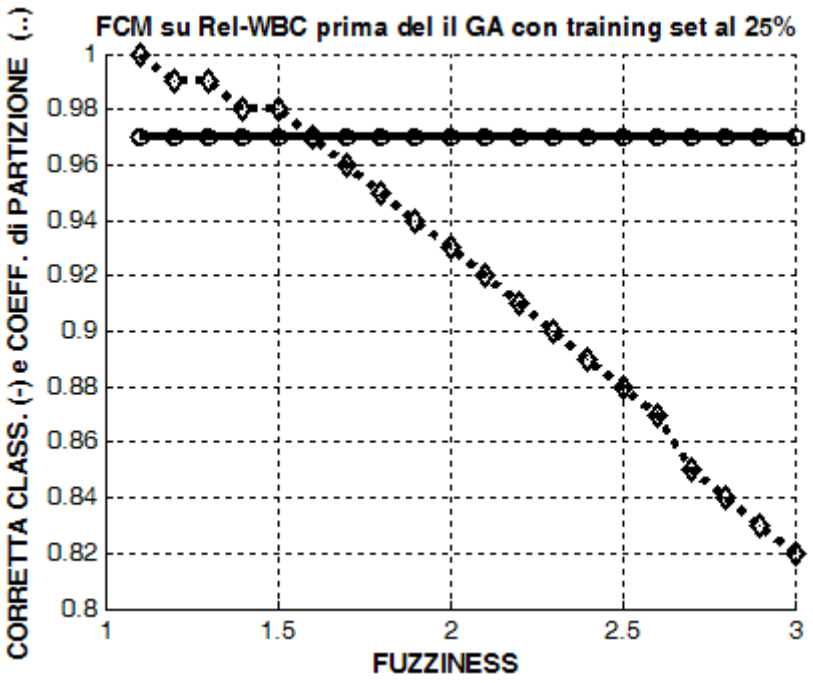

figura 50: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 25%

```
Risultati ROPTICS:
Impossibile trovare il numero di cluster specificato
```

Training set: 25% dopo il GA

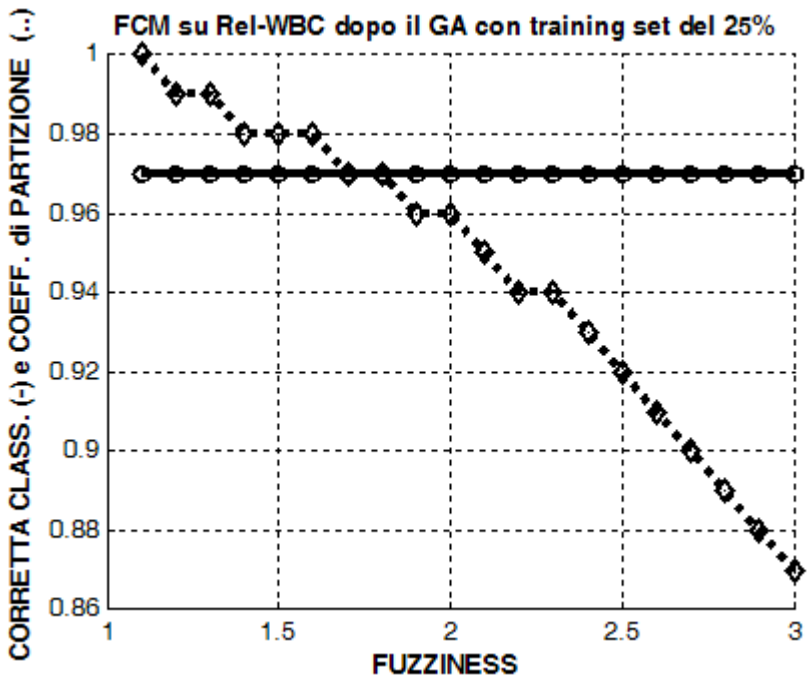

figura 51: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 25%

```
Risultati ROPTICS:
Tasso di classificazione: 95.75%
```

La tabella che segue riassume i risultati ottenuti.

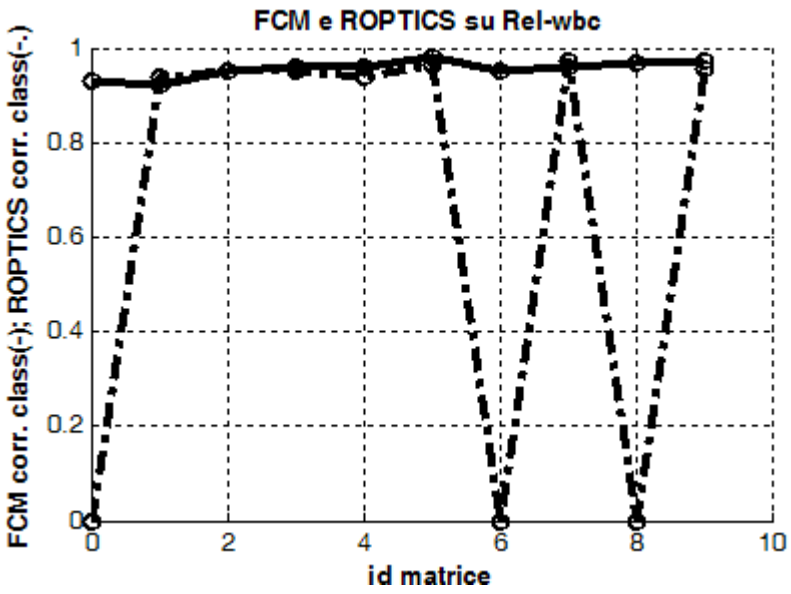

figura 52: Andamento del tasso di classificazione di FCM per m=2 (linea continua) e del tasso di classificazione di ROPTICS (punto-linea). I valori sull'asse delle ascisse rappresentano il numero della matrice usata

Test effettuato su relational cc

Per i test che seguono ROPTICS ha i seguenti parametri:

```
minPts: 20
ξ: 0.0001
```

Training set: 5% prima del GA

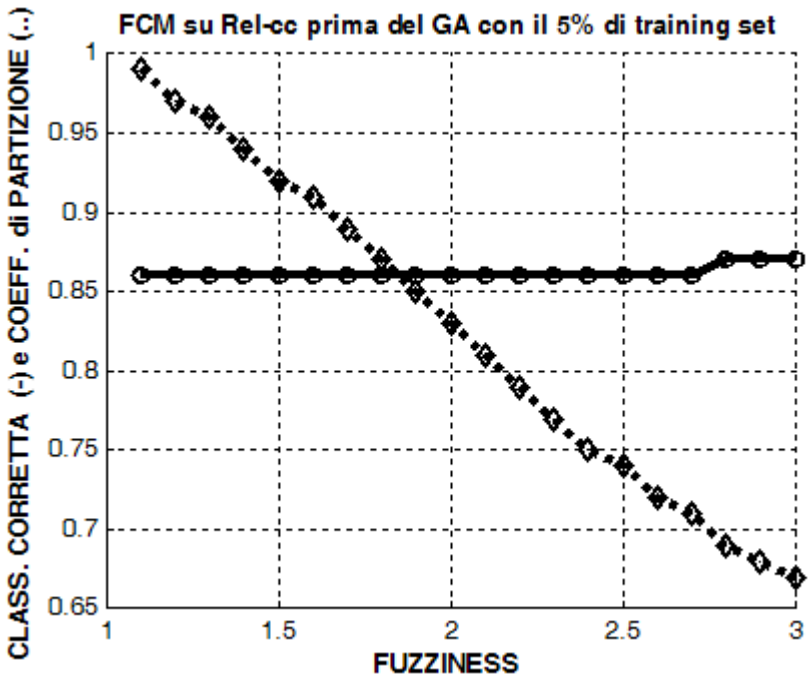

figura 53: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 5%

```
Risultati ROPTICS:
Tasso di classificazione: 84.44%
```

Training set: 5% dopo il GA

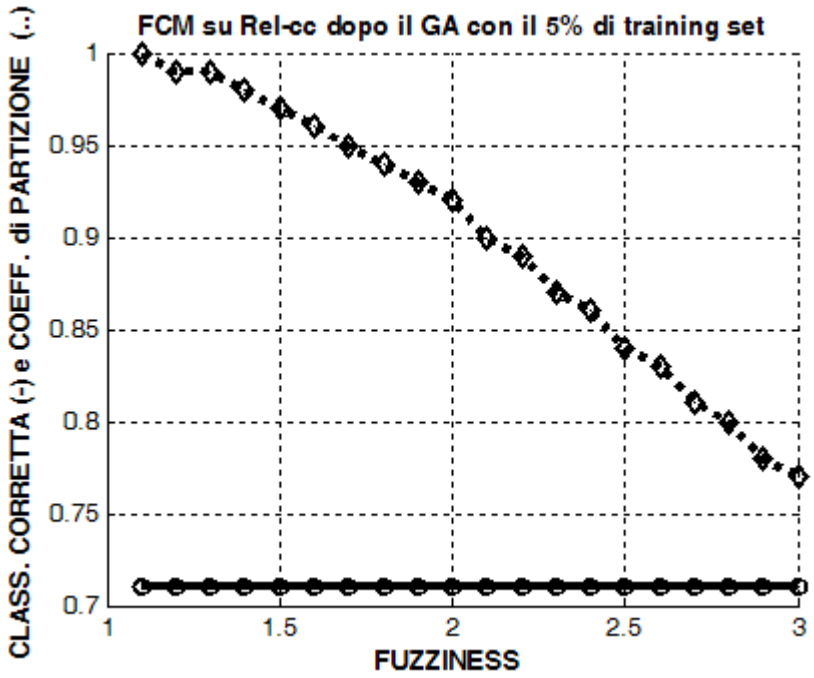

figura 54: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 5%

```
Risultati ROPTICS:
Tasso di classificazione: 73.88%
```

Training set: 10% prima del GA

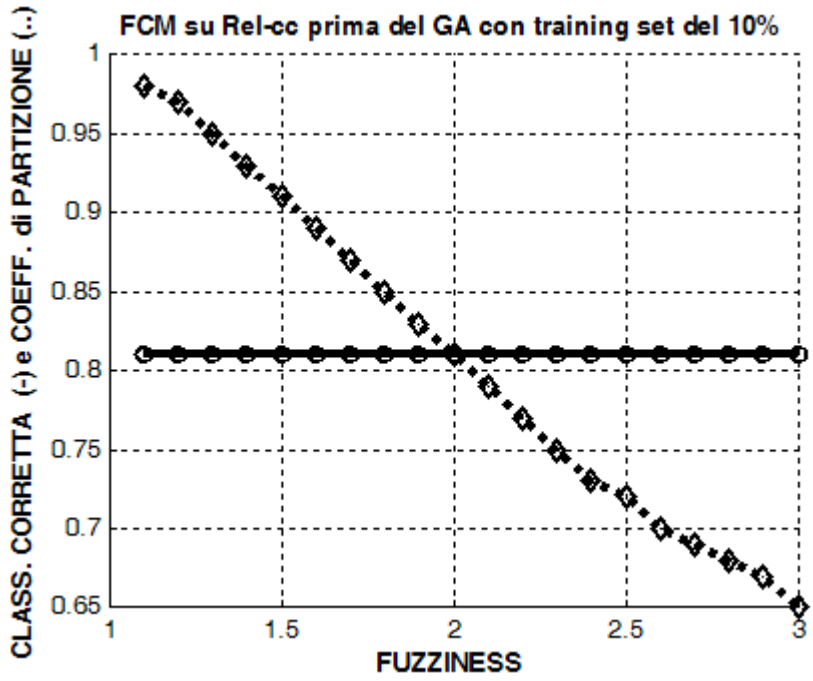

figura 55: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 10%

Risultati ROPTICS:

Tasso di classificazione: 30%

Training set: 10% dopo il GA

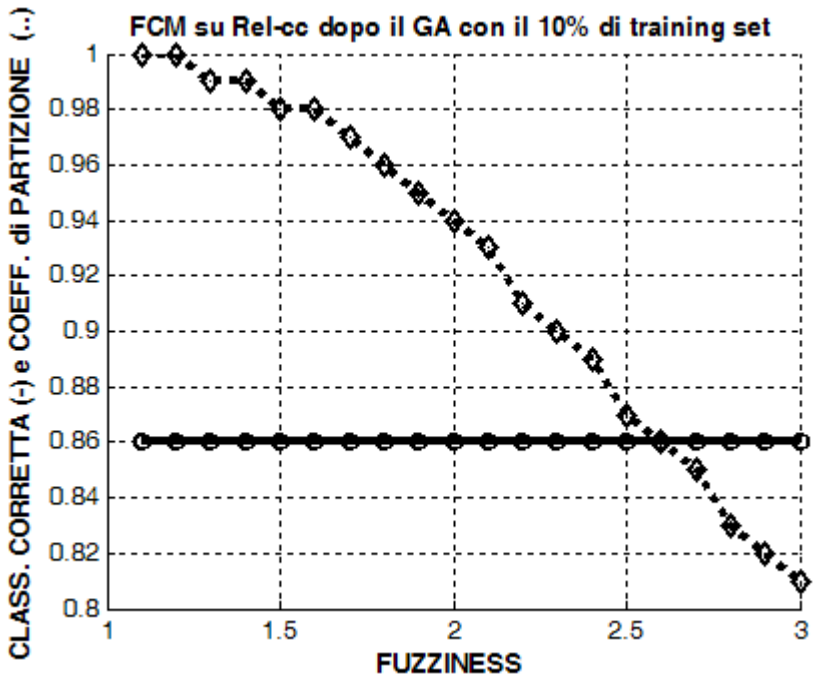

figura 56: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 10%

```
Risultati ROPTICS:
Tasso di classificazione: 84.44%
```

Training set: 15% prima del GA

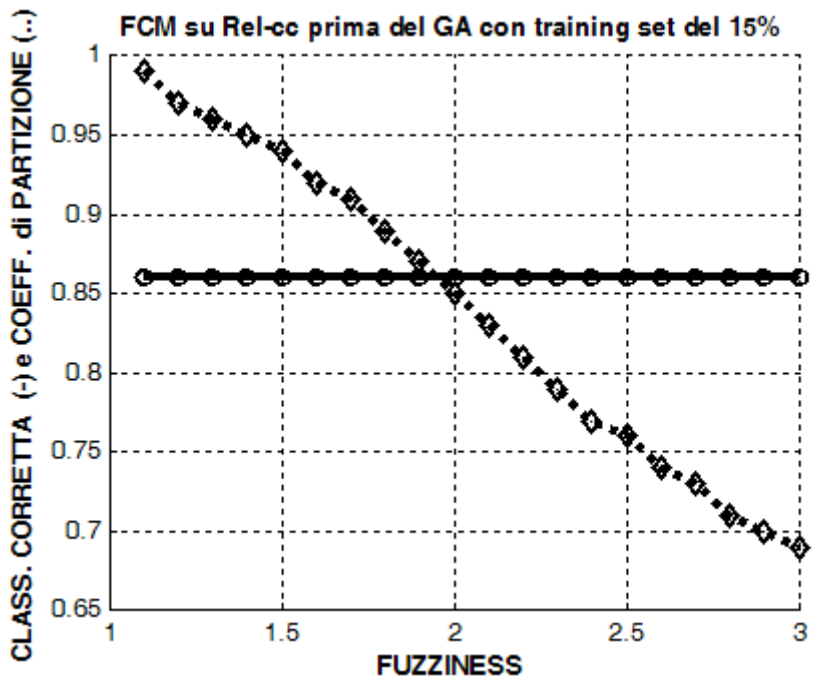

figura 57: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 15%

```
Risultati ROPTICS:
Tasso di classificazione: 81.66%
```

Training set: 15% dopo il GA

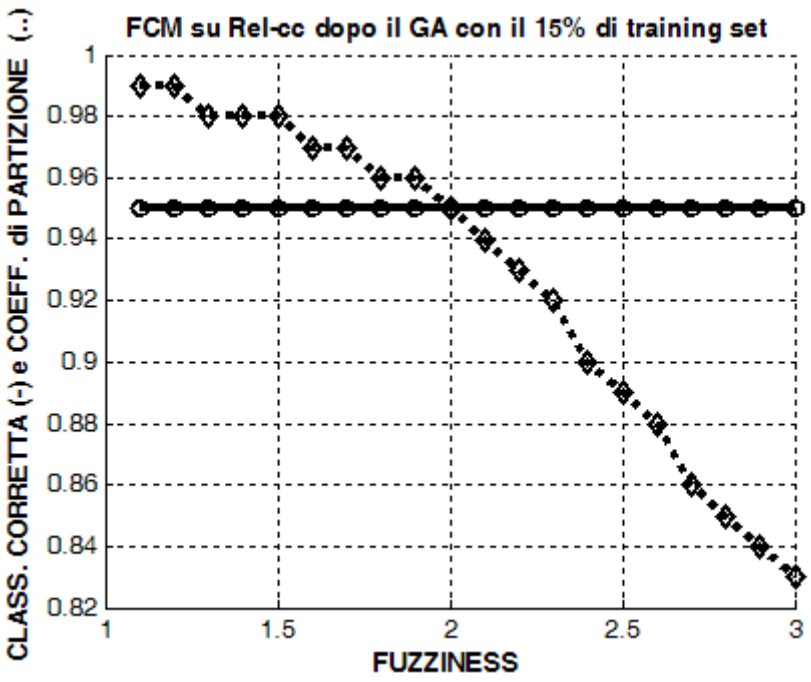

figura 58: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 15%

```
Risultati ROPTICS:
Tasso di classificazione: 88.88%
```

Training set: 20% prima del GA

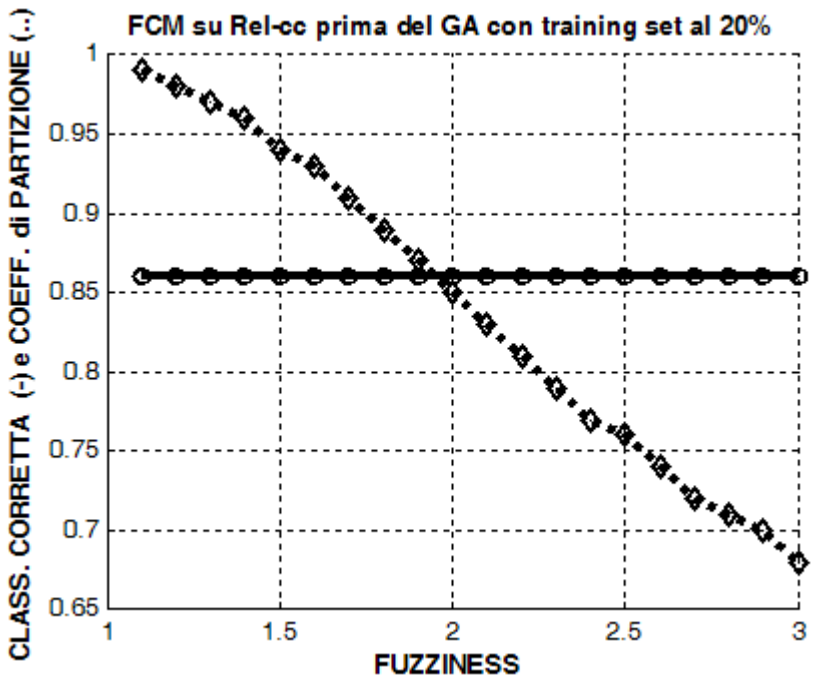

figura 59: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 20%

```
Risultati ROPTICS:
Tasso di classificazione: 84.44%
```

Training set: 20% dopo il GA

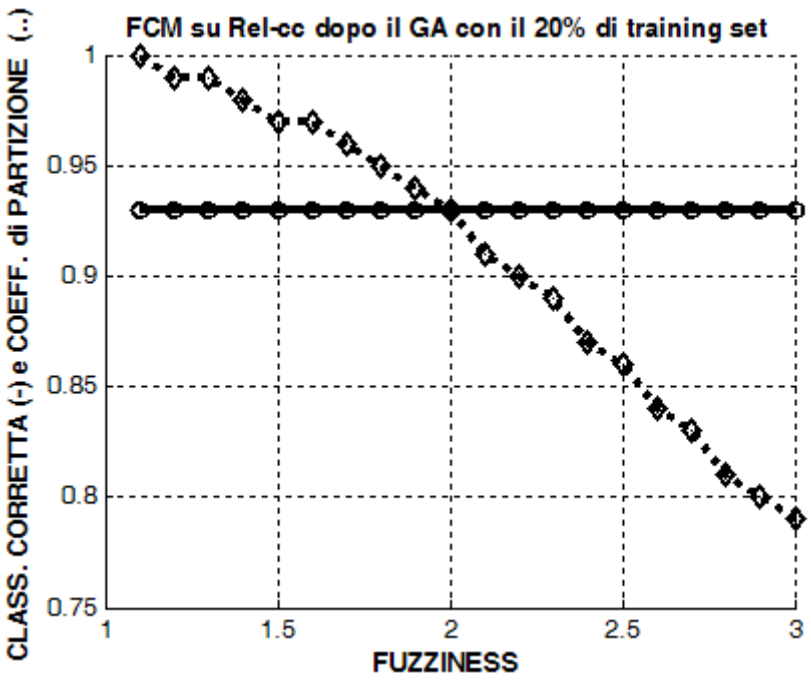

figura 60: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 20%

```
Risultati ROPTICS:
Tasso di classificazione: 93.33%
```

Training set: 25% prima del GA

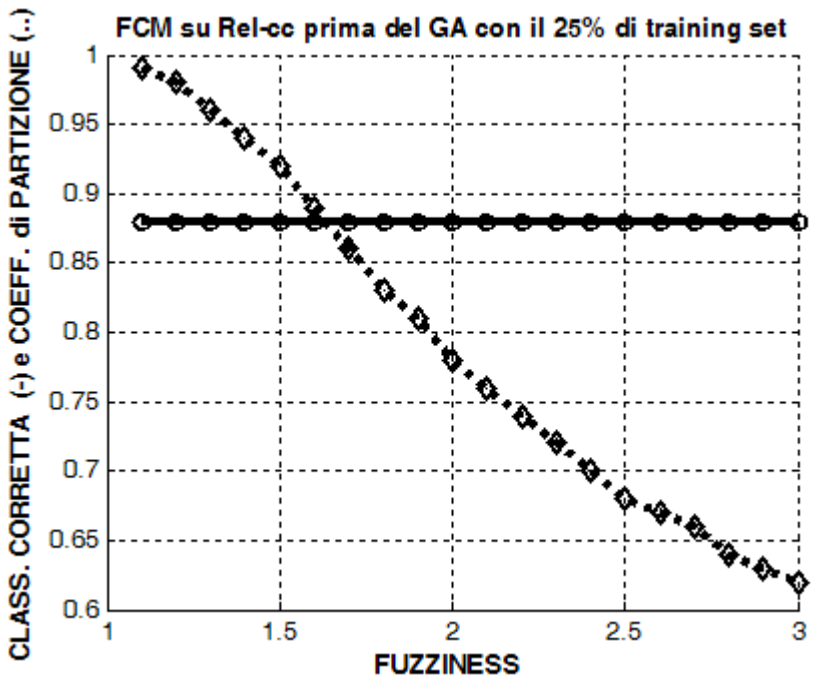

figura 61: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta prima del GA con training set al 25%

```
Risultati ROPTICS:
Tasso di classificazione: 81.66%
```

Training set: 25% dopo il GA

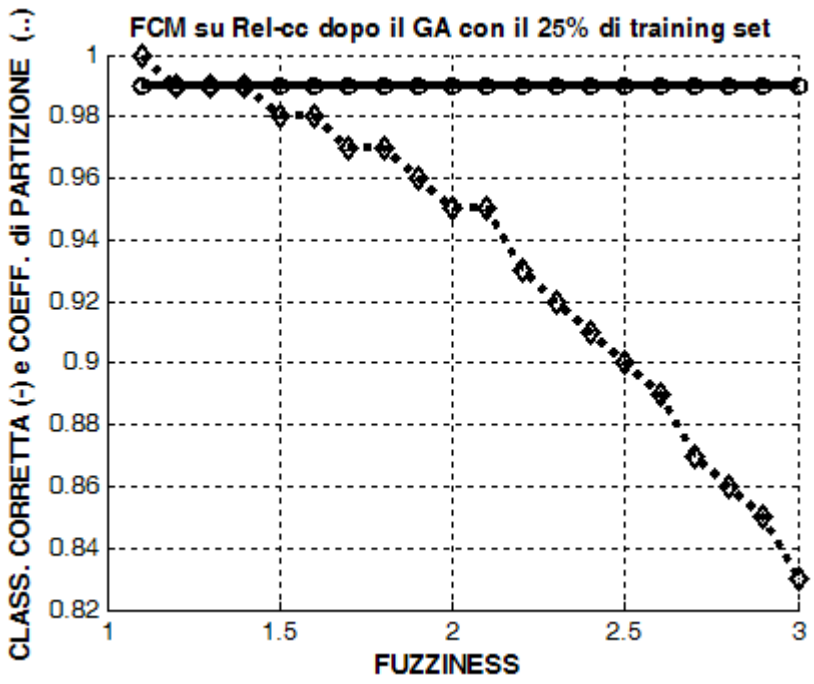

figura 62: Andamento del tasso di classificazione e del coefficiente di partizione di FCM al variare di *m* con matrice ottenuta dopo il GA con training set al 25%

```
Risultati ROPTICS:
Tasso di classificazione: 96.66%
```

Si consideri anche il seguente grafico che riporta il confronto tra il tasso di classificazione di FCM e ROPTICS al variare della percentuale di training (per FCM si pone m=2).

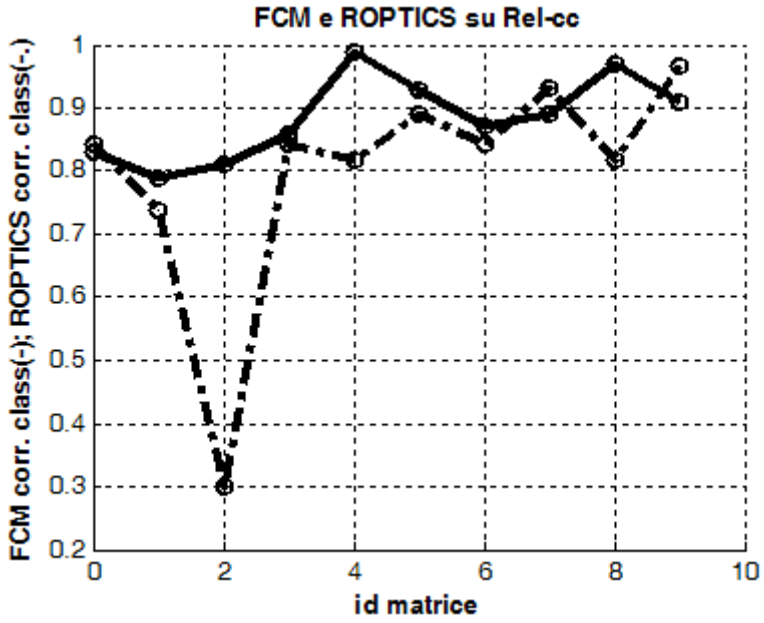

figura 63: Andamento del tasso di classificazione di FCM per *m*=2 (linea continua) e del tasso di classificazione di ROPTICS (punto-linea). I valori sull'asse delle ascisse rappresentano il numero della matrice usata

Al fine di illustrare la tematica dell'alta dimensionalità si consideri un ultimo esempio; l'insieme di dati seguente è costituito da tre anelli di punti in $\mathbb{R}^3$; questi anelli sono allineati sull'asse z e sono così rappresentabili:

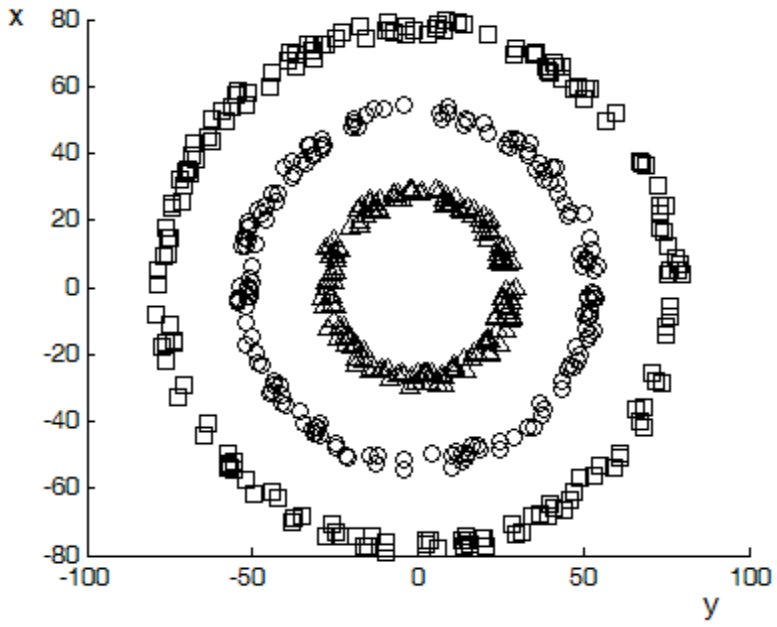

figura 64: Insieme di dati costituito da tre 'anelli' proiettato sul piano XY

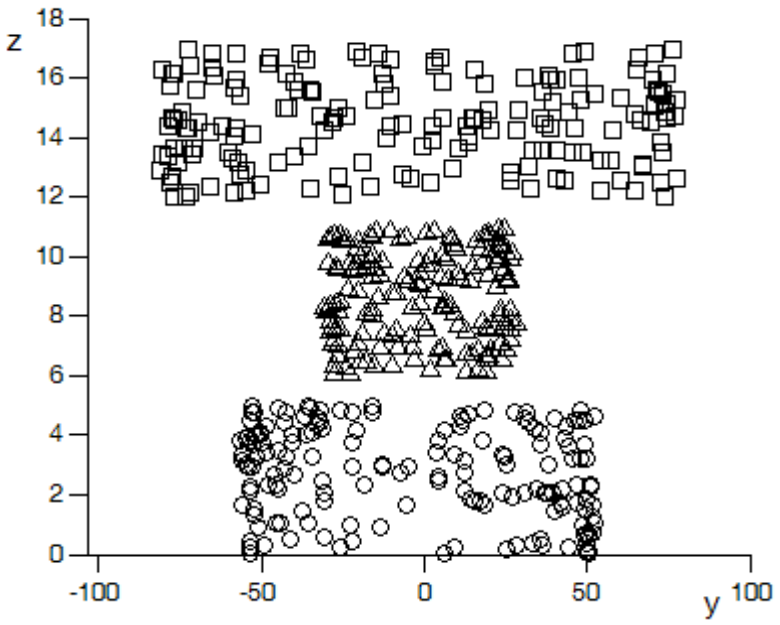

figura 65: Insieme di dati formato da tre 'anelli' proiettato sul piano YZ

Le figure precedenti constano di 500 pattern in totale equamente distribuiti tra i tre cluster (che hanno, quindi, diverse densità); in questo caso l'esecuzione di FCM produce i risultati che seguono:

```
numero pattern = 500
m = 2
tasso di classificazione = 51.6%
coefficiente di partizione = 0.7266
```

Aumentando il numero di punti a mille si evidenzia il calo di prestazioni.

```
numero pattern = 1000
m = 2
tasso di classificazione = 35.8%
coefficiente di partizione = 0.517
```

In realtà si nota che se i tre cluster hanno un diverso numero di punti e identiche densità il tasso di classificazione si attesta attorno al 35% e non varia passando da 500 a 1000 pattern. Per quanto riguarda ROPTICS, per l'esempio riportato nelle figura precedenti, non si ottiene la separazione dei cluster; mentre per i tre cluster a eguale densità si ottengono i risultati seguenti:

```
numero pattern = 1000
minPts = 65
ξ = 0.0001
tasso di classificazione = 65.8%
```

Il tasso di classificazione si mantiene inalterato anche per 500 pattern.

7.3 Confronto con NE-FRC

L'inconveniente maggiore nell'uso di questo algoritmo è rappresentato dalla non convergenza. Questo problema è già evidente con insieme di dati piuttosto piccoli. Si consideri ad esempio l'insieme raffigurato di seguito, contenente 150 elementi:

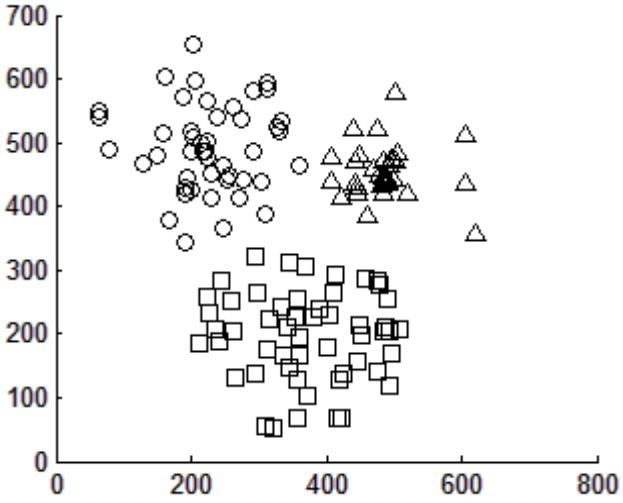

figura 66: Insieme di 150 elementi organizzati in tre cluster sparsi

Le prove sono state effettuate con i parametri riportati:

```
Caratteristiche dell'insieme di dati e parametri:
numero pattern: 150
cluster:3
λ: 1000
numero massimo di iterazioni: 1000
Criterio di arresto: 0.0001
```

Come si osserva da prove sperimentali se c'è convergenza il tasso di riconoscimento si mantiene molto alto, si noti che con *m*=1.1 in genere si ha convergenza. Il grafico riportato di seguito sintetizza i risultati di dieci prove effettuate per i valori di *m* da 1.1 a 3. Si è potuto constatare che aumentando la fuzziness aumenta la probabilità di non avere convergenza entro un numero ragionevole di iterazioni.

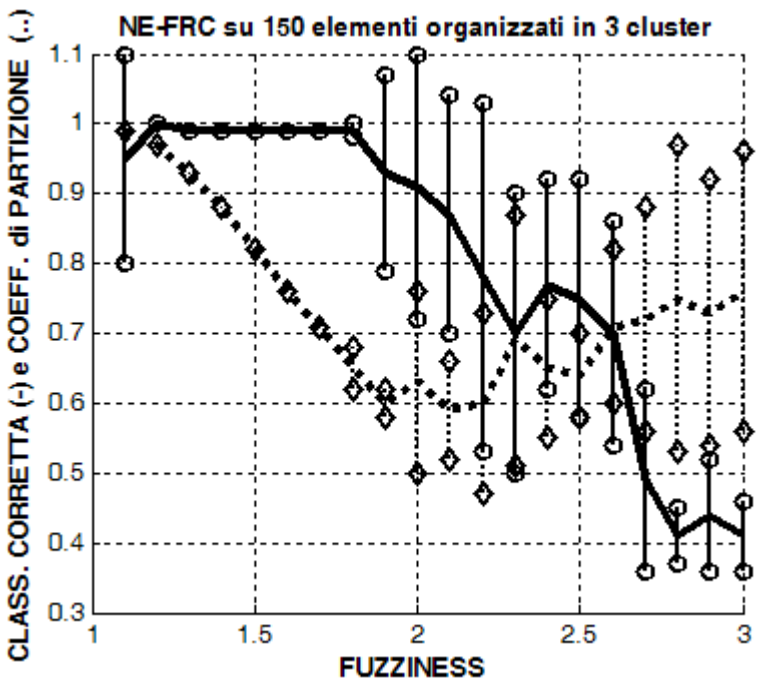

figura 67: Andamento del tasso di classificazione medio (-) e del coefficiente di partizione (...)

Aumentando il numero di punti si vede che per *m* basso il tasso di classificazione si mantiene alto fino ad un insieme di almeno 500 punti. La complessità di NE-FRC lo rende inutilizzabile per insiemi più ampi. Inoltre anche il coefficiente di partizione scende all'aumentare della fuzziness, ciò significa che la classificazione è più incerta.
Seguono i dati relativi all'applicazione di ROPTICS:

```
minPts: 10
ξ: 0.0001
Tasso di classificazione: 95.33%
```

Si noti, comunque, che l'esecuzione di ROPTICS su insiemi di dati piccoli non produce sempre buoni risultati.
Per quanto riguarda l'insieme di dati relational cc i risultati per 10 esecuzioni

dell'algoritmo sono riportati nel grafico seguente:

```
Caratteristiche dell'insieme e parametri:
Numero pattern: 180
Cluster:2
λ: 1000
Numero massimo di iterazioni: 999
Criterio di arresto: 0.0001
```

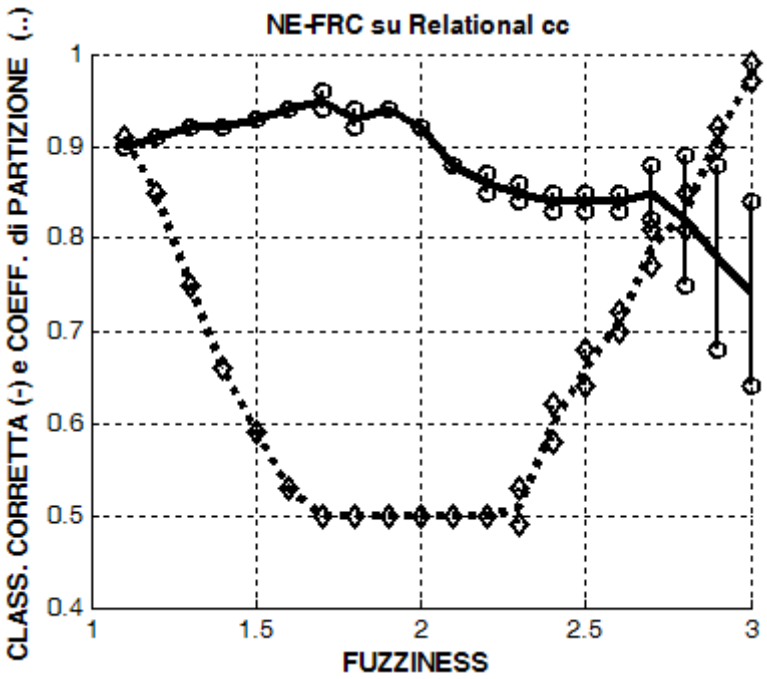

figura 68: Esecuzione di NE-FRC sul relational cc

Come si è detto la corretta classificazione di ROPTICS sfiora il 90.55%.

Per quanto riguarda la matrice iris, si è utilizzata quella presentata al capitolo 3

```
Caratteristiche dei dati e parametri:
Numero pattern: 150
Cluster:3
λ: 1000
Numero massimo di iterazioni: 1000
Criterio di arresto: 0.0001
```

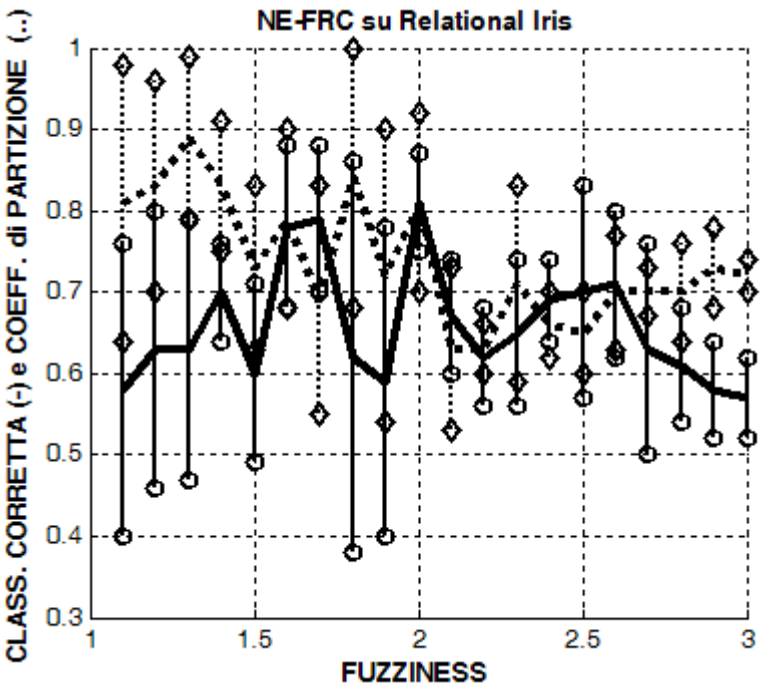

figura 69: Esecuzione di NE-FRC sul relational Iris

Date le sue dimensioni non è stato possibile utilizzare il set breast cancer.

7.4 Confronto con altri algoritmi

Gli algoritmi che sono stati utilizzati sono AP, FNM, ARCA, NeRFCM, FCMdd, ReRFCM (Relational Robust FCM); verranno proposti vari test e si valuteranno i risultati di ognuno dei metodi oggetto di valutazione. Le varie prove sono state effettuate 4 volte per ognuno dei valori della fuzziness seguenti: 1.1 1.5 2.0 2.5 3.0 e con un termine di accuratezza che vale 0.0001.

Il primi test riguarda ancora iris; supponiamo di avere la matrice non euclidea rappresentata in figura 17.

Test sull'insieme R-iris.

Test effettuato con ARCA:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

Il grafico seguente riassume i risultati ottenuti; da notare che per lo stesso parametro ARCA ha dato sempre lo stesso risultato.

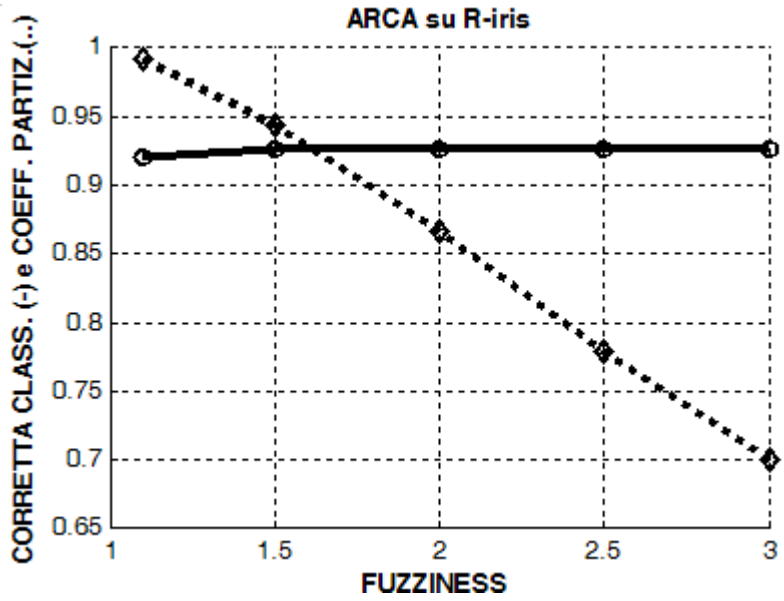

figura 70: Risultati di ARCA

Test effettuato con AP:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

Con AP non è possibile distinguere i cluster; ne vengono rilevati solo due.

Test effettuato con ReRFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

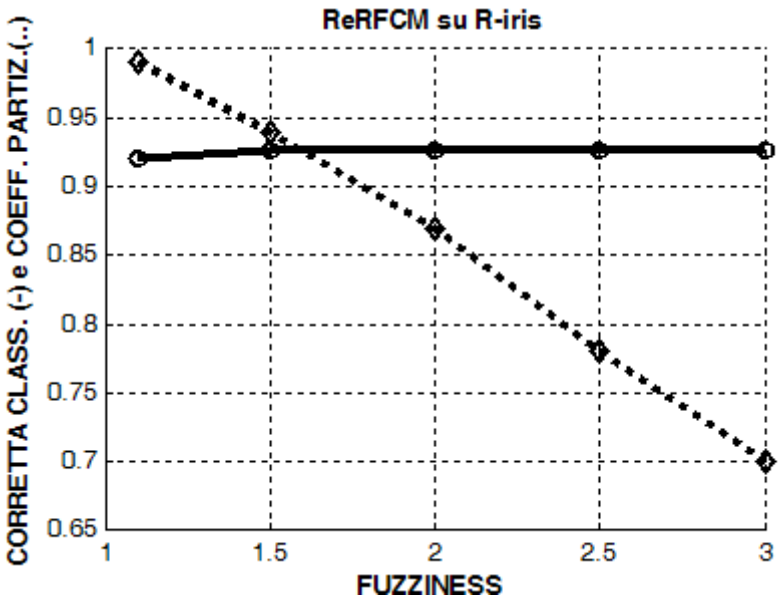

figura 71: Risultati di ReRFCM

Come si vede il grafico è praticamente identico a quello di ARCA.

Test effettuato con NeRFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

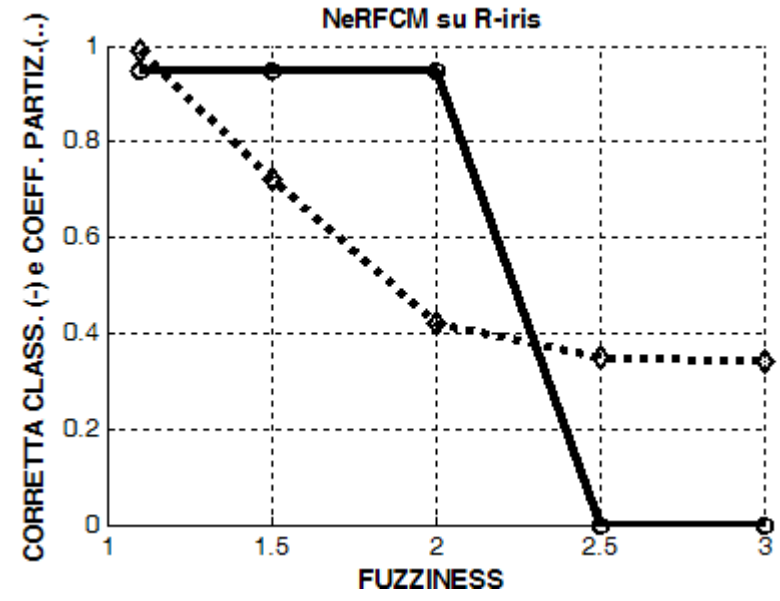

figura 72: Risultati di NeRFCM

Il tasso di classificazione si mantiene al 94.6% fino ad m = 2; oltre questo valore NERFCM non riesce a separare i tre cluster.

Test effettuato con FNM

Non è stato possibile far convergere l'algoritmo.

Test effettuato con FCMdd

Non è stato possibile trovare i parametri giusti per far terminare l'algoritmo correttamente.

Test su relational cc

Test effettuato su ARCA:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

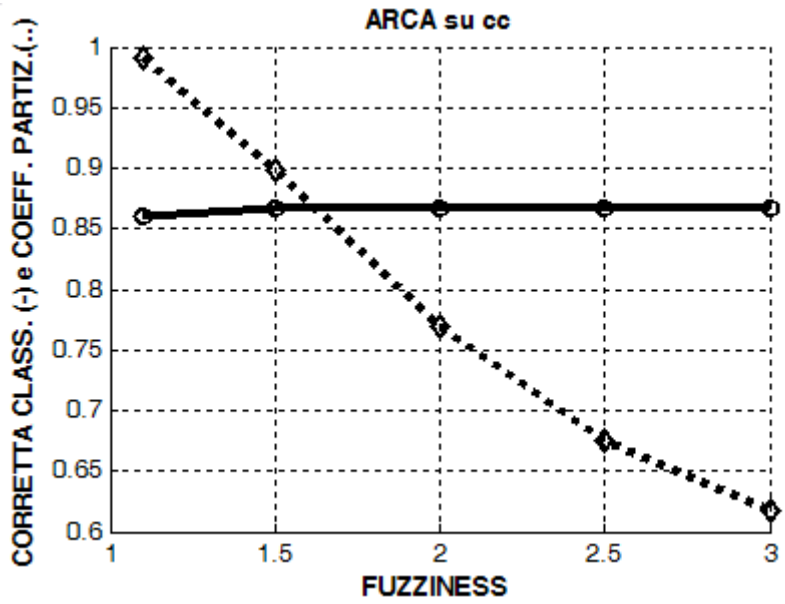

figura 73: Risultati di ARCA

Test effettuato su AP:

Tramite AP non è possibile distinguere i cluster.

Test effettuato con ReFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

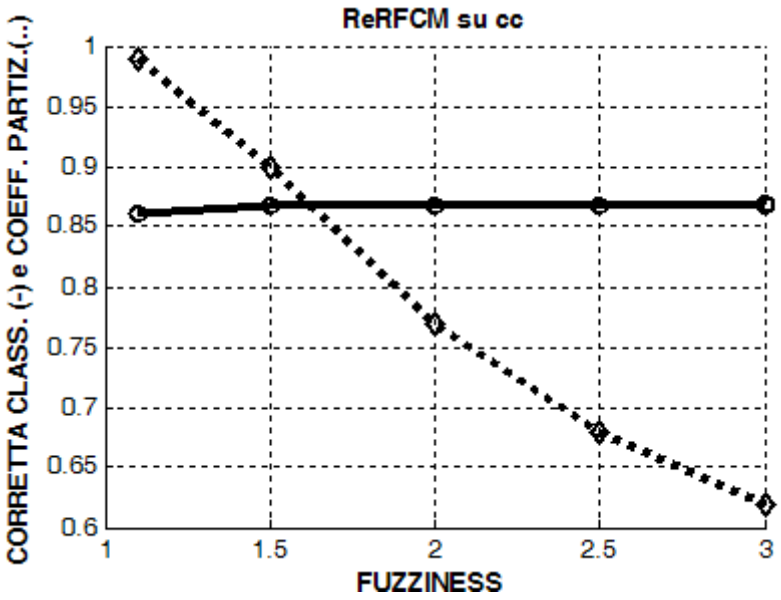

figura 74: Risultati di ReRFCM

Test effettuato con NeRFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

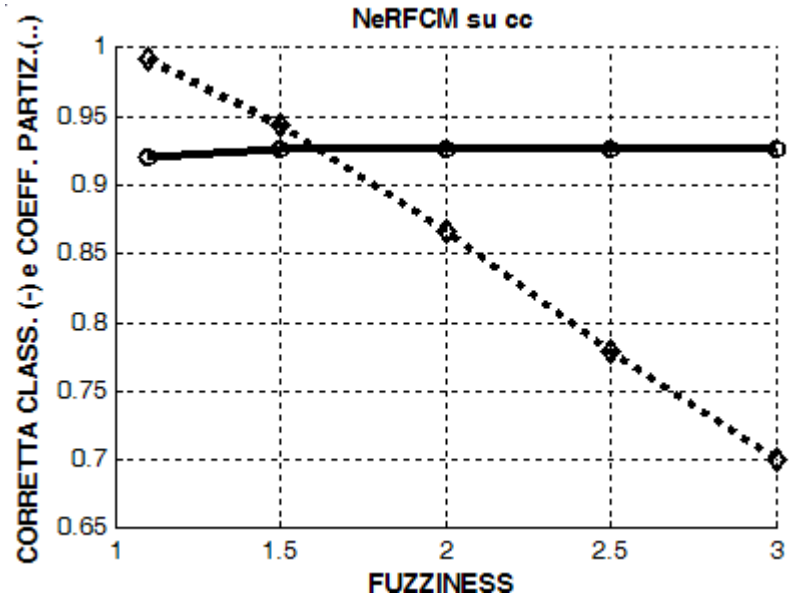

figura 75: Risultati di NeRFCM

Andare oltre ad m=2 comporta l'impossibilità di separare i cluster.

Test effettuato con FNM

Non è stato possibile far convergere l'algoritmo.

Test effettuato con FCMdd

```
Inizializzazione medoidi = personalizzata
Partizione iniziale: casuale
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 100
Ignore level: 179
Subset cardinality: 177
```

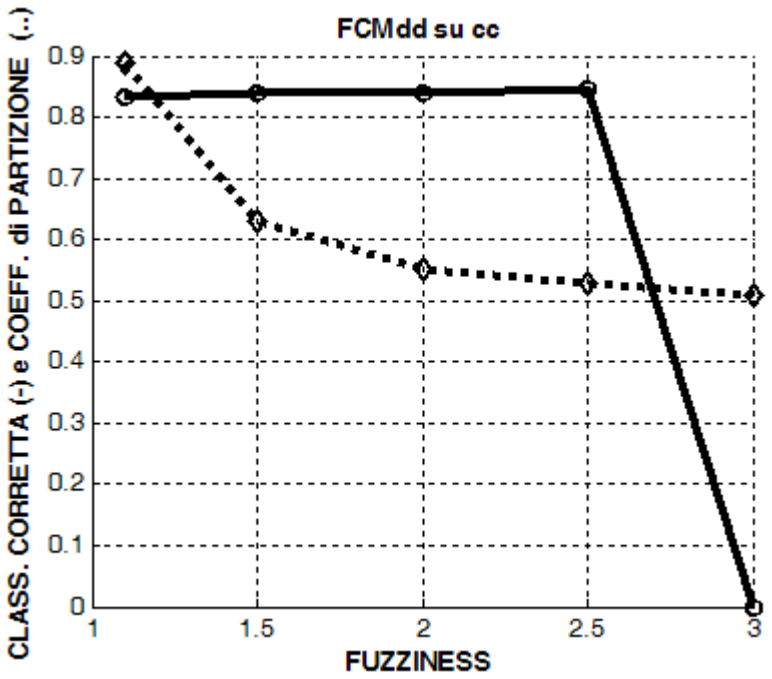

figura 76: Risultati di FCMdd

Test sul data set Winsconsin breast cancer

Test effettuato con ARCA:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

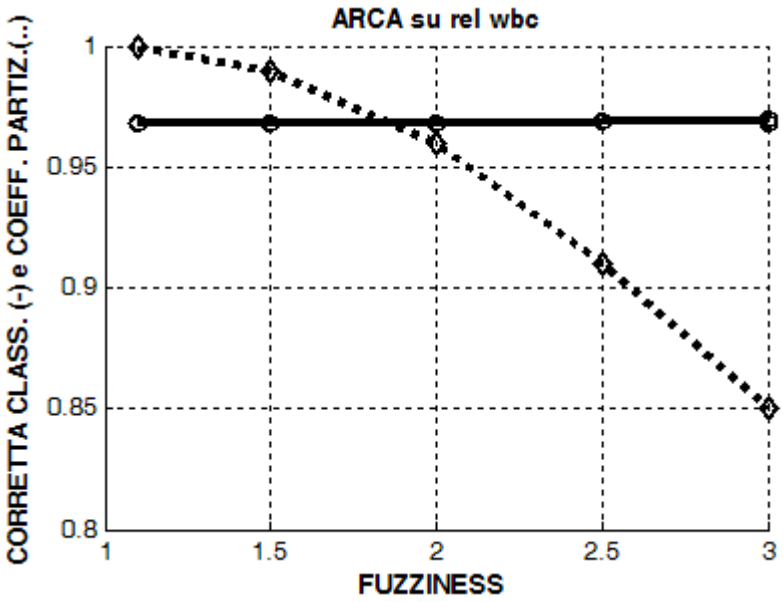

figura 77: Risultati di ARCA

Prove effettuate su AP:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

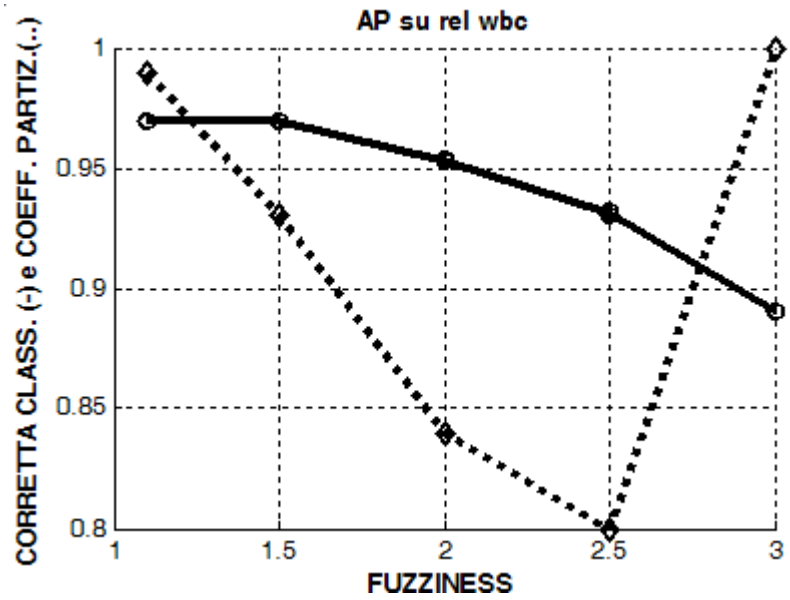

figura 78: Risultati di AP

Prove effuttuate su ReRFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

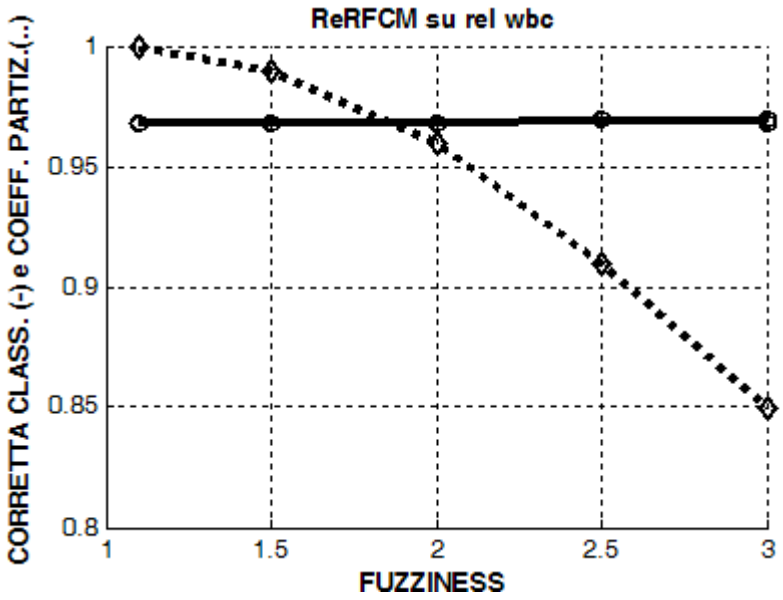

figura 79: Risultati di ReRFCM

Prove effettuate su NeRFCM:

```
Partizione iniziale: Random
Termine di accuratezza: 0.0001
Massimo numero di iterazioni 300
```

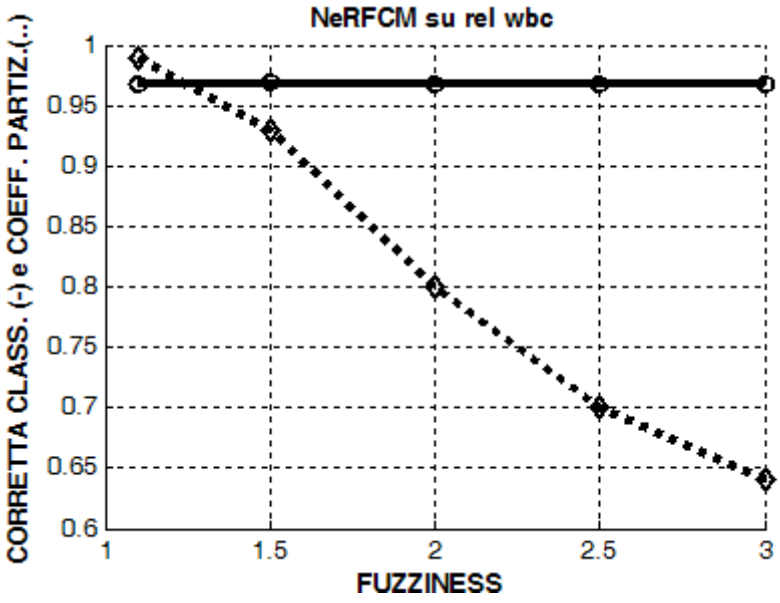

figura 80: Risultati di NeRFCM

Prove effettuate su FNM:

Non è stato possibile far convergere l'algoritmo

Prove effettuate con FCMdd:

Non è stato possibile far terminare correttamente l'algoritmo.

7.5 Considerazioni riassuntive

Dagli esempi riportati appare chiaro come ogni algoritmo abbia il suo ambito applicativo specifico; inoltre nel prendere atto delle prestazioni in termini di efficienza non ci si può esimere da considerazioni relative alle prestazioni ed alle limitazioni a cui i dati devono sottostare per poter far funzionare l'algoritmo. Ad esempio l'FNM non sembra funzionare in alcun caso ma occorre notare che il suo dominio applicativo è esteso anche a dati senza proprietà metriche (vincolo necessario per altri algoritmi). In altri casi la complicata messa a punto dei parametri rende inservibili alcune implementazioni nei casi pratici (non si conoscono a priori i parametri ottimali e, non avendo la corretta classificazione, non è semplice cercarli).

In sintesi:

1. Il problema di NeRFCM è il rapido degrado di prestazioni all'aumentare della fuzziness; il coefficiente di partizione, cioè, diminuisce velocemente.
2. ARCA risulta equivalente all'FCM.
3. L'algoritmo NE-FRC utilizzato risulta non scalabile, inoltre presenta un palese degrado di efficacia qualora si intenda generare una partizione fuzzy (ovvero il coefficiente di partizione scende all'aumentare della fuzziness in modo molto marcato).
4. FNM non risolve mai i cluster in maniera ottimale (si ferma in punti di ottimo locale)
5. AP spesso non distingue tutti i cluster
6. Per dataset sferici uguali ed equidistanti FCM è il migliore
7. La messa a punto dei parametri di FCMdd non è banale

Come si è visto non esiste una soluzione migliore di un'altra. Il fatto che FCM sia stato il migliore ai fini dei confronti non lo rende migliore ovunque, in particolare un grosso limite è costituito dall'impossibilità di gestire agglomerati di forma non convessa, inoltre occorre conoscere in anticipo il numero di cluster che si intende trovare. Questo discorso vale anche per gli altri.

Per quanto concerne il sistema di estrazione delle somiglianze si osserva che ROPTICS

ha prestazioni paragonabili a FCM soprattutto per percentuali alte di training set.

Capitolo 8

Conclusioni e sviluppi futuri

Uno dei problemi principali degli algoritmi relazionali è rappresentato dal fatto che hanno una convergenza garantita solo la dissimilarità è caratterizzata da proprietà metriche. Gli algoritmi che, invece, sono adatti a dati non metrici presentano tempi di convergenza enormi (NE-FRC) o addirittura, in molti casi, non convergono (FNM). Per contro, algoritmi oggetto applicati allo spazio relazionale (ARCA o FCM) sono robusti e convergono sempre, ma per alta dimensionalità soffrono dei problemi dovuti all'uso di distanze. Utilizzando il concetto di clustering basato sulla densità, nella presente tesi è stato progettato e sviluppato ROPTICS. Tale algoritmo non necessita della definizione di una metrica sullo spazio di rappresentazione, ma viene eseguito soltanto sulla base delle somiglianze espresse nella matrice relazionale. ROPTICS non soffre quindi del problema dell'alta dimensionalità (degli algoritmi-oggetto) e della non convergenza (degli algoritmi relazionali classici) quindi costituisce una importante ed inesplorata alternativa nel trattamento di dati relazionali. Le principali limitazioni di ROPTICS sono relative a insiemi di dati con cluster a parziale sovrapposizione ed alla robustezza nei confronti del rumore.

Il primo problema è stato parzialmente risolto individuando i punti “ibridi” come quelli a norma inferiore ad una certa soglia, facendo in modo che tali punti non impediscano di distinguere i diversi cluster. Un ulteriore sviluppo consiste nell'uso di operatori più accurati della norma.

Per quanto riguarda il secondo problema per gestire il rumore (composto dai pattern detti outlier) è sufficiente abbassare il raggio massimo del vicinato e modificare l'algoritmo in modo da segnare i punti con distanze di raggiungibilità infinita ed assegnarli, al termine, ad un cluster di rumore (si noti che OPTICS è robusto al rumore, nella presente tesi non

ci si è posti il problema di rendere ROPTICS altrettanto robusto).

Un'altra questione rilevante riguarda la difficoltà a ricostruire la corretta struttura dei cluster (si veda la figura 21); questo limite potrebbe essere ridotto introducendo la logica fuzzy per rendere il clustering più robusto, progettando e sviluppando un metodo per identificare una partizione fuzzy dei dati che tenga conto dell'informazione di densità racchiusa nel "grafico di raggiungibilità", per ridurre la dipendenza del clustering density based dai parametri.

Il grande limite degli algoritmi classici riguarda la necessità di conoscere in anticipo il numero di cluster da trovare. Sebbene ROPTICS sia stato strutturato per la ricerca di un numero di cluster stabilito dall'utente può essere modificato per la scoperta automatica di tale numero. Al termine dell'analisi del grafico delle distanze di raggiungibilità, ROPTICS crea una struttura ad albero in cui ogni nodo rappresenta un cluster. Per scoprire il numero ottimale di cluster si può definire una misura di qualità del clustering (analoga all'indice di qualità definito al capitolo 3); si calcola tale indice per ogni nodo dell'albero e si scelgono i nodi (cioè i cluster) con indice superiore ad una certa soglia (che può essere specificata dall'utente).

Appendice A

Ulteriori dettagli implementativi

Oltre allo studio in termini teorici, riveste grande importanza l'attività di sviluppo. Uno degli aspetti che hanno influenzato questa attività riguarda la necessità di rendere conforme l'applicazione ad alcune convenzioni usate nell'ambito applicativo in cui è stato sviluppato il lavoro; ad esempio il formato dei file di ingresso/uscita è stato reso uguale a quello delle applicazioni esistenti utilizzate per effettuare i test.

A.1 Analisi dei requisiti

Anzitutto il programma leggerà le matrici da file di ingresso in modalità testo (sintassi Matlab™); ai fini della valutazione del suo funzionamento è possibile inserire nel file di ingresso il vettore contenente il numero del cluster a cui appartiene il pattern corrispondente. A tale proposito un aspetto assume particolare importanza; supponiamo di avere un vettore che contiene per ogni pattern il cluster di appartenenza e il risultato dell'algoritmo, occorrerà fare in modo da trovare la giusta corrispondenza tra le etichette.

Si noti che l'algoritmo non è studiato per essere robusto al rumore; in fase di preparazione del insieme di dati si possono, comunque, assegnare i punti di rumore all'ultimo cluster e, indicando questo fatto tramite una opzione apposita, non si tiene conto di questi punti ai fini del calcolo del tasso di classificazione. Per poter effettuare più esecuzioni e tenere traccia del risultato si ha la possibilità di utilizzare un file di log in cui vengono scritti i risultati di elaborazioni successive.

A.2 Sintassi e regole di funzionamento

Per specificare i parametri ed i dati in ingresso si ricorre ad opzioni a riga di comando. La

forma del comando è la seguente

```
optics -n num [-m num][-e num][-c num][-1 num] [-g num][-h][-v][-z]
[-r][-l] <nomefile>
```

Il significato delle opzioni è descritto di seguito:

-m Specifica il parametro *minPts*; se non viene specificato, tale parametro viene posto a 20 per default

-e Specifica il parametro ε; se non viene specificato direttamente nel caso in cui viene creato il file delle reachability-distance (opzione **-z** assente) esso corrisponde al massimo elemento della matrice, altrimenti (per velocizzare il procedimento) viene posto uguale al massimo numero rappresentabile dal calcolatore.

-c Serve ad indicare il parametro ξ, se omesso viene posto a 0.00001

-1 Indica la norma minima del vettore rappresentante il pattern per considerare tale punto valido ai fini della ricerca dei vicini; ciò vale a dire che, se un pattern supera in norma tale valore, viene considerato un border object.

-n Definisce la dimensione della matrice di ingresso

-g Indica il numero di cluster che si vuole ottenere (deve essere equivalente al numero di cluster presenti nel vettore delle appartenenze in ingresso per poter consentire il calcolo del tasso di classificazione)

-h Produce un clustering gerarchico (equivalente relazionale di OPTICS); in tal caso viene stampata a video la struttura gerarchica.

-v Stampa informazioni aggiuntive

-z Non crea il file delle reachability-distance

-l Scrive un file di log con i risultati della elaborazione

Il formato del file di ingresso deve essere come nel seguente esempio (formato Matlab™):

```
R=[
      0     1     1     8     7;
      1     0     2     6     5;
      1     2     0     6     1;
      8     6     6     0     6;
      7     5     1     6     0;
];
ccPrev=[
1;
1;
1;
2;
2;
];
```

Notare che i nomi (R e ccPrev) non sono importanti; inoltre, non è necessario che la prima matrice sia simmetrica.

Ad ogni elaborazione vengono prodotti diversi file. Il principale di essi è **`ErrclassROPTICS.m`** contenente i vettori di appartenenza dei cluster (prima la classificazione corretta fornita in ingresso e poi quella prodotta da ROPTICS), e la matrice di classificazione. Nel caso in cui l'opzione ***-z*** non sia presente, viene creato il file **`reach.m`** che può essere usato per fare il grafico delle reachability-distance tramite Matlab™; il file contiene, inoltre, la corrispondenza tra il numero d'ordine iniziale dei pattern ed il numero d'ordine che proviene dall'esecuzione del ciclo a comune con OPTICS. Se è attiva l'opzione -l viene aperto in append il file **`statROPTICS.txt`** contenente le seguenti informazioni:

`t-begin, t-end, minPts, csi, tasso classificazione`

ossia il tempo di inizio e di fine dell'elaborazione, il *minPts*, lo ξ, ed il tasso di classificazione. L'uso di questo file è stato introdotto per permettere di eseguire diverse volte ROPTICS tramite file batch.

A.3 Descrizione delle classi

Di seguito è riportato il diagramma delle classi nel linguaggio semiformale UML. Per ognuna di esse sarà fornita la descrizione ed il codice sorgente.

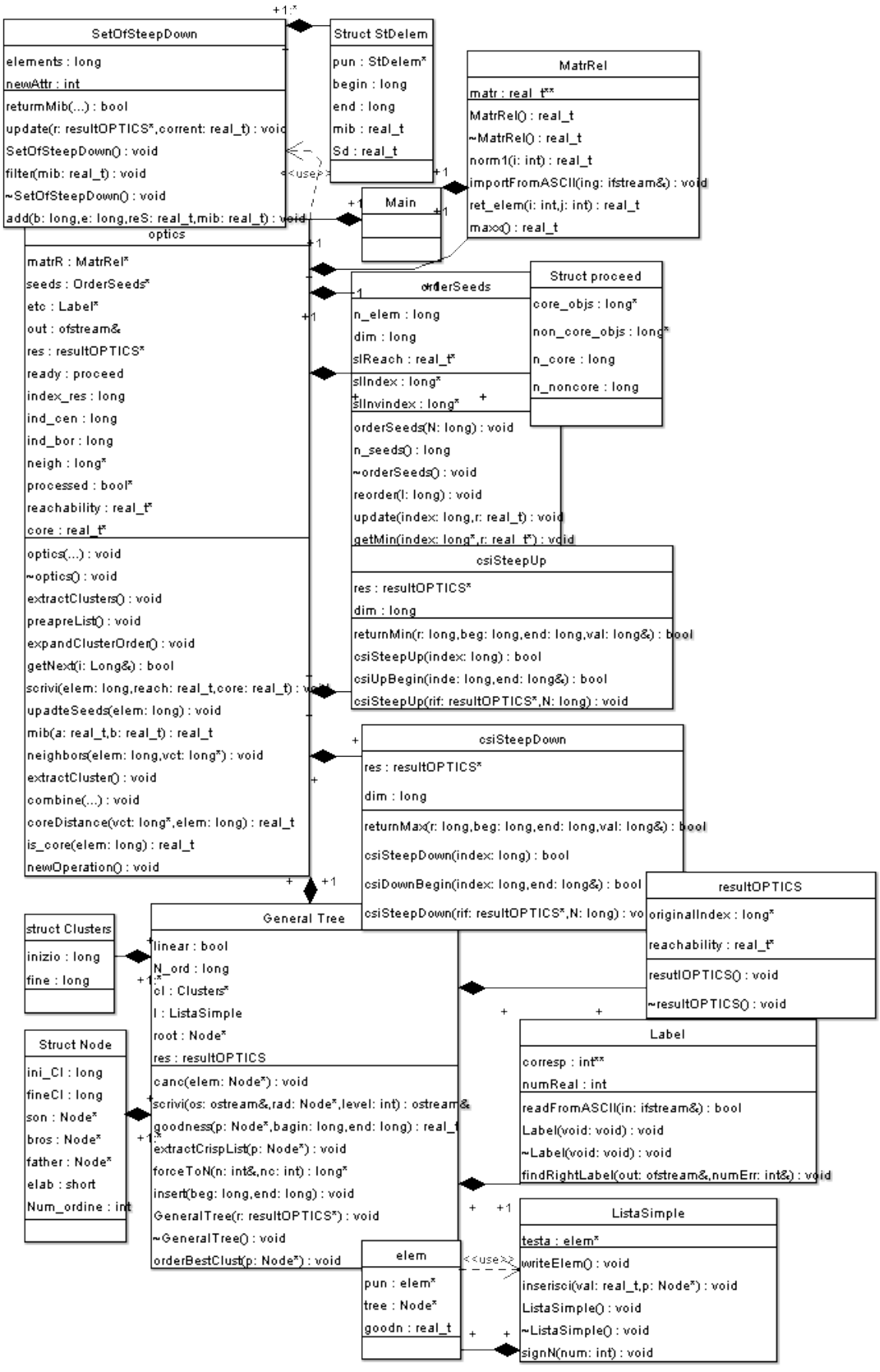

figura 81: Diagramma delle classi

Appendice B

Codice sorgente

Viene riportato il codice sorgente di tutti i moduli che costituiscono l'applicazione con i commenti relativi. L'applicazione è stata sviluppata in linguaggio C++ tramite Microsoft™ Visual Studio .NET.

```
////////////////////////////////////////////////////////////////////////////////
////
// main.cpp

//Davide Bolognesi 17-1-2006
//

#include "Label.h"
#include "MatrRel.h"
#include "globalconst.h"
#include "globalfun.h"
#include "globaltype.h"

#include <string.h>
#include <stdlib.h>

#define OUTPUT "errclassROPTICS.m"
#define OUTPUTOPTICS "reach.m"
#define LOGFILE "statROPTICS.txt"

#include <fstream>
using namespace std;

extern const str_t bar;
extern real_t eps;
extern int minPts;
extern real_t csi;
extern const str_t prgTitle;
extern const str_t usrErr0;
extern const str_t usrErr1;
extern char beginTime[9]={'1'};
extern const str_t usrErr2;
extern const str_t usrUsage;
extern const str_t usrFileErr;
```

```
extern const str_t done;
extern const str_t end;
extern const str_t logIni;
extern const str_t usrErrDim;
extern const str_t usrErrCsi;
extern const str_t usrErrNorm;
extern const str_t usrErrNCl;
extern const str_t prgFindEps;
extern void msgExit(const str_t usrErr, int nErr);
extern inline void beep(int);
extern void displayTime();

extern int numErr=0;
extern long N=0;
extern int N_Clust=-1;
extern bool matl=true;
extern bool infEps=true;
extern bool verbose=false;
extern bool logfile=false;
extern bool norm1=false;
extern real_t norm1min=0;
extern bool gerarchico=false;
extern bool lastClustNoise=false;

void graspOpt(int argc,char* argv[]){ /*(1)*/
      for(int j=1;j<argc-1;){
            char optType;
            if(argv[j][0]=='-'){
                  optType=argv[j][1];
            }
            else{
                  msgExit(usrUsage,0);
            }
            switch(optType){
            case 'e':
                  eps=(real_t)atof(argv[j+1]);
                  infEps=false;
                  j+=2;
                  break;
            case 'm':
                  minPts=atoi(argv[j+1]);
                  j+=2;
                  break;
            case 'n':
                  N=atoi(argv[j+1]);
                  j+=2;
                  break;
            case 'c':
                  csi=(real_t)atof(argv[j+1]);
                  if(csi>1){
                        msgExit(usrErrCsi,2);
                  }
                  j+=2;
                  break;
            case 'h':
                  gerarchico=true;
                  verbose=true;
                  j+=1;
                  break;
            case 'g':
                  N_Clust=atoi(argv[j+1]);
```

```
                eps=INFINITY/3;
                if(N_Clust<=0){
                        msgExit(usrErrNCl,2);
                }
                j+=2;
                break;
        case 'l':
                logfile=true;
                j++;
                break;
        case 'r':
                lastClustNoise=true;
                j+=1;
                break;
        case 'v':
                j+=1;
                verbose=true;
                break;
        case '1':
                norm1min=(real_t)atof(argv[j+1]);
                if(norm1min<=0){
                        msgExit(usrErrNorm,2);
                }
                j+=2;
                norm1=true;
                break;
        case 'z':
                j+=1;
                matl=false;
                break;
        default:
                msgExit(usrErr2,2);
        }
    }
}

void main( int argc, char* argv[] )
{
    char *in_file;
    if (argc<3){
        msgExit(usrErr1,1);
    }
    else{
        cout<<endl<<bar<<endl;
        displayTime();
        cout<<prgTitle<<endl;
        graspOpt(argc,argv);
        if(N<=0){
            msgExit(usrErrDim,1);
        }
        MatrRel mRel;
        in_file=argv[argc-1];
        ifstream ing;
        ing.open(in_file,ios::in);
        if(!ing.is_open()) {
            msgExit(usrFileErr,0);
        }
        if(!mRel.importFromASCII(ing)){
            msgExit(usrFileErr,0);
        }
        Label etc;
```

```
etc.readFromASCII(ing);
ing.close();
if(verbose){
	displayTime();
	cout<<done<<endl;
}
if(matl){//(2)
	if(eps==INFINITY/3){
		displayTime();
		cout<<prgFindEps<<" ";
		eps=mRel.maxx();
		cout<<done<<endl;
	}
}
_strtime(beginTime);
ofstream usc,out;
resultOPTICS r_opt;
if(matl){/*(2)*/
	usc.open(OUTPUTOPTICS,ios::out);
	if(!usc.is_open()){
		msgExit(usrFileErr,0);
	}
	usc<<"Rd=["<<endl;
}
optics opt(&mRel, usc ,&r_opt, &etc);
if(matl){
	usc<<"];"<<endl;
	usc<<"\n%Corrispondenza con l'ordine originale\n";
	usc<<"Corresp=[\n";
	for(int i=0;i<N;i++){
		usc<<r_opt.originalIndex[i]+1<<" ";
	}
	usc<<"\n];";
}
usc.close();
opt.extractClusters();
displayTime();
cout<<end<<endl;
if(N_Clust!=-1){
	out.open(OUTPUT,ios::out);
	if(!out.is_open()){
		msgExit(usrFileErr,0);
	}
	etc.findRightLabel(out,numErr);
	out.close();
	if(logfile){
		ofstream usc;
		ifstream in;
		in.open(LOGFILE,ios::in);
		bool fl=false;
		if(!in){
			fl=true;
		}
		in.close();
		usc.open(LOGFILE,ios::out|ios::app);
		if(fl){
			usc<<logIni<<"\n";
		 }
		usc<<beginTime<<"  ";
		char endTime[9];
		_strtime(endTime);
```

```
                        usc<<endTime<<"\t   "<<minPts<<'\t'<<csi<<"\t      "<<1-
(float)numErr/N<<endl;
                        usc.close();
                  }
            }
      }
      cout<<bar<<endl;
      beep(1);
}

void writeErr(){
                  ofstream usc;
                  ifstream in;
                  in.open(LOGFILE,ios::in);
                  bool fl=false;
                  if(!in){
                        fl=true;
                  }
                  in.close();
                  usc.open(LOGFILE,ios::out|ios::app);
                  if(fl){
                        usc<<logIni<<"\n";
                  }
                  usc<<beginTime<<"  ";
                  char endTime[9];
                  _strtime(endTime);
                  usc<<endTime<<"\t   "<<minPts<<'\t'<<csi<<"\t
"<<"0"<<endl;
                  usc.close();
}

/*
Questo modulo gestisce l'interazione con l'utente
(1) Funzione che acquisisce i parametri a riga di comando, se si vuole
interfacciare
o integrare ROPTICS con un'altra applicazione occorre modificarla.
(2) Se si è specificato che si desidera produrre il file delle reachability-
distance
si fa in modo che la visualizzazione tramite Matlab si più chiara (cioè si
abbassa il
valore di infinito in modo da non vedere il grafico troppo schiacciato).
Il file creato prende il nome di reach.m (lo si può cambiare nella #define in
alto);
la procedura Matlab che disegna il grafico è la seguente:

clear reach;
reach;
bar(1:size(Rech,1),Rech(1:size(Rech,1)),'g');


*/

////////////////////////////////////////////////////////////////////////////
////
// optics.h

//Davide Bolognesi 8-1-2006
//
class GeneralTree;

#pragma once
```

```
#include <fstream>
using namespace std;

#include "MatrRel.h"
#include "globaltype.h"
#include "GeneralTree.h"
#include "Label.h"
#include "globalconst.h"
#include "orderSeeds.h"
#include "SetOfSteepDown.h"
#include "csiSteepDown.h"
#include "csiSteepUp.h"
#include "resultOPTICS.h"

struct proceed{                 /*(1)*/
      long* core_objs;
      long* non_core_objs;
      long n_core;
      long n_noncore;
};

class optics                    /*(2)*/
{
      friend void scriviAppartenenza(const optics&,GeneralTree&,ostream&);
private:
      const MatrRel * matrR;
      OrderSeeds* seeds;
      Label* etc;
      ofstream& out;
      resultOPTICS* res;
      proceed ready;
      long index_res;
      long ind_cen;
      long ind_bord;
      long* neigh;
      bool* processed;
      real_t* reachability;
      real_t* core;

      void neighbors(long,long*);
      real_t coreDistance(long*,long);
      bool is_core(long);
      inline real_t mib(real_t,real_t);
      bool getNext(long&);
      void prepareList();
      void
combine(SetOfSteepDown&,GeneralTree&,long,long,csiSteepDown,csiSteepUp);
public:
      optics(const MatrRel*, ofstream&,resultOPTICS*,Label*);
      void extractClusters();
      void expandClusterOrder(void);
      void updateSeeds(long);
      void scrivi(long,real_t,real_t);
      ~optics(void);
};
/*
(1) La struttura contiene la suddivisione tra core point e border object
(2) La classe costituisce il nucleo dell'applicazione; tutte le elaborazioni
sono
svolte a partire dai metodi che la classe fornisce
*/
```

```
////////////////////////////////////////////////////////////////////////////
////
// optics.cpp

//Davide Bolognesi 17-2-2006
//
#include ".\optics.h"

#define INFINITO 2*(1/(1-csi))*eps /*(1)*/
#define UNDEF -1
#define END -1


extern real_t csi=real_t(.00001);
extern real_t eps=INFINITY/3; /*(1)*/
extern int minPts=20;
extern bound=0;

extern long N;
extern bool matl;
extern bool verbose;
extern bool norm1;
extern int N_Clust;
extern bool infEps;
extern real_t norm1min;
extern const str_t done;
extern const str_t prgFR;
extern const str_t prgFC;
extern const str_t prgUnnest;
extern const str_t prgPreOrd;
extern const str_t prgStrCl;
extern const str_t prgStrCl2;

extern void printslash(int);
extern void printpercent(int,int);

void optics::neighbors(long elem,long* vct){
      long i=0;
      long j=0;
      if(norm1 && matrR->norm1(elem)<norm1min){/*(3)*/
            vct[0]=END;
            return;
      }
      for(;j<N;j++){
            if(j!=elem){
                  if(matrR->ret_elem(elem,j)<=eps){
                        vct[i]=j;
                        i++;
                  }
            }
      }
      vct[i]=END;
}

optics::optics(const MatrRel* M, ofstream& usc,resultOPTICS* rso, Label*
l):out(usc)
{                 /*(6)*/
      matrR=M;
      res=rso;
      etc=l;
```

```
	index_res=0;
	seeds=new OrderSeeds(N);
	processed=new bool[N];
	reachability=new real_t[N];
	neigh = new long[N];
	core=new real_t[N];
	for(long ind=0;ind<N;ind++){
		processed[ind]=false;
		reachability[ind]=INFINITO;
	}
	displayTime();
	cout<<prgPreOrd<<endl;
	prepareList();
	displayTime();
	cout<<prgFR<<endl;
	expandClusterOrder();
}

void optics::updateSeeds(long elem){/*(4)*/
	real_t c_dist=coreDistance(neigh,elem);
	real_t new_r_dist;
	for(long ind=0;ind<N;ind++){
		if(neigh[ind]==END){break;}
		new_r_dist=max(c_dist,matrR->ret_elem(elem,neigh[ind]));
		seeds->update(neigh[ind],new_r_dist);
	}
}

void optics::scrivi(long elem,real_t reach,real_t core){
	res->originalIndex[index_res]=elem;
	res->reachability[index_res]=reach;
	if(matl){
		out<<reach<<" ;"<<endl;
	}
	printpercent(index_res,N);
	index_res++;
}

real_t optics::coreDistance(long* vect,long elem){
	int len=0;
	for(;len<N;len++){
		if(vect[len]==END){break;}
	}
	if(len<minPts){
		return INFINITO;
	}
	else{
		real_t eps1=INFINITO;
		long num;
		real_t eps2;
		for(long i=0;i<N;i++){
			if(i==elem){continue;}
			num=-1;
			eps2=matrR->ret_elem(elem,i);

			for(long j=0;j<N;j++){
				if(matrR->ret_elem(elem,j)<=eps2){
					num++;
				}
			}
			if(num>=minPts){
```

```
                        if(eps2<eps1){
                              eps1=eps2;
                        }
                  }
            }
            return eps1;
      }
}

void optics::expandClusterOrder(void){
      long w;
      while(getNext(w)){
            if(processed[w]==false){
                  neighbors(w,neigh);
                  processed[w]=true;
                  reachability[w]=INFINITO;
                  core[w]=coreDistance(neigh,w);
                  scrivi(w,reachability[w],core[w]);
                  if(core[w]!=INFINITO){
                        updateSeeds(w);
                        }
                        long index;
                        real_t value;
                        while(seeds->n_seeds()!=0){
                        seeds->getMin(&index,&value);
                        if(processed[index]!=true){
                              reachability[index]=value;
                              neighbors(index,neigh);
                              processed[index]=true;
                              core[index]=coreDistance(neigh,index);
                              scrivi(index,reachability[index],core[index]);
                              if(core[index]!=INFINITO){
                                    updateSeeds(index);
                              }
                        }
                        }
            }
      }
      cout<<"\r";
}

optics::~optics(void)
{
      delete[] processed;
      delete[] neigh;
      delete[] core;
      delete[] reachability;
}

inline real_t optics::mib(real_t a,real_t b){
            return max(a,b);
}

void optics::extractClusters(){           /*(7)*/
      displayTime();
      cout<<prgFC<<endl;
      csiSteepDown cD(res,N);
      csiSteepUp cU(res,N);
      SetOfSteepDown steepD;
      GeneralTree tree(res);
      long num_steep=0;
```

```
	long index=0;
	long fine;
	real_t mib=0;	/*(8)*/
	while(index<N){
		mib=max(mib,res->reachability[index]);
		steepD.filter(mib);
		if(cD.csiDownBegin(index,fine)){
			steepD.update(res,res->reachability[index]);
			steepD.filter(mib);
			steepD.add(index,fine,res->reachability[index],0);
			index=fine;
			mib=res->reachability[index];
		}
		else{
			if(cU.csiUpBegin(index,fine)){
				steepD.update(res,res->reachability[index]);
				steepD.filter(mib);
				if(index<N-1){
					mib=res->reachability[index];
					combine(steepD,tree,index,fine,cD,cU);
				}
				else{
					combine(steepD,tree,index,index+1,cD,cU);
					break;
				}
				index=fine;

			}
			else{
		    index++;
			}
		}
	}
	if(verbose){
	displayTime();
	  cout<<done<<endl;
	  cout<<prgStrCl<<endl<<tree;
	}
	if(N_Clust>-1){
		int x=0;
		long* partizione;
	  partizione=tree.forceToN(x,N_Clust);
		for(int y = 0;y < N; y++){
			etc->corresp[1][res->originalIndex[y]]=partizione[y];
		}
		if(partizione){
			delete[] partizione;
		}
		if(verbose){
			displayTime();
			cout<<done<<endl;
			cout<<prgStrCl2<<endl<<tree;
		}
	}
}

bool optics::is_core(long s){
	int count=0;
	for(int in=0;in<N;in++){
		if(in!=s){
			if(matrR->ret_elem(s,in)<=eps){
```

```
                        count++;
                        if(count>=minPts){return true;}
                  }
            }
      }
      return false;
}
void optics::prepareList(){   /*(2)*/
      ready.core_objs=new long[N];
      ready.n_core=0;
      ready.non_core_objs=new long[N];
      ready.n_noncore=0;
      if(infEps){
            for(int x=0;x<N;x++){
                  ready.core_objs[ready.n_core++]=x;
            }
      }
      else{
            for(int x=0;x<N;x++){
                  printpercent(x,N);
                  if(is_core(x)){
                        ready.core_objs[ready.n_core++]=x;
                  }
                  else{
                        ready.non_core_objs[ready.n_noncore++]=x;
                  }
            }
      }
      ind_cen=0;
      ind_bord=0;
      cout<<"\r";
}

bool optics::getNext(long& c){
      if(ready.n_core!=0){
            c=ready.core_objs[ind_cen++];
            --ready.n_core;
            return true;
      }
      if(ready.n_noncore!=0){
            c=ready.non_core_objs[ind_bord++];
            --ready.n_noncore;
            return true;
      }
      return false;
}

void optics::combine(SetOfSteepDown& steepD,GeneralTree& t,long iniCsiUp,long
fineCsiUp,csiSteepDown d, csiSteepUp u){
                  real_t mibD=0;/*(5)*/
                  long i=0;
                  long reachEnd=0;
                  long fineClust=0;
                  long inizioClust=0;
                  long begin=0;
                  long end=0;
                  while(steepD.returnMib(i,mibD,begin,end)){      //(9)
                real_t reachStart=res->reachability[begin];
                real_t reachEnd=res->reachability[fineCsiUp];
                        if(res->reachability[fineCsiUp]*(1-csi)>=mibD){
                              if(reachStart*(1-csi)>=reachEnd){
```

```
                                    long val;

      if(!d.returnMax(fineCsiUp,begin,end,val)){i++;continue;}
                                    inizioClust=val;  //(10)
                                    fineClust=fineCsiUp-1;
                              }
                              else if(reachEnd*(1-csi)>=reachStart){
                                    long val;

      if(!u.returnMin(iniCsiUp,begin,end,val)){i++;continue;}
                                    fineClust=val;    //(11)
                                    inizioClust=begin;
                              }
                              else{ //(12)
                                    fineClust=fineCsiUp-1;
                                    inizioClust=begin;
                              }
                              if(fineClust-inizioClust>=minPts){
                              t.insert(inizioClust,fineClust);
                              }
                        }
                        i++;
                  }
}

/*

(1) Sono valori sufficientemente alti (non è importante il valore in sè)
tali che INFINITO vale più di eps (che concettualmente vale a sua volta
infinito);
è necessario per i calcoli.
(2) Si esegue solo quando si specifica un certo eps; serve a distinguere i
border objects
dai core objects; infatti se non lo faccio quando elaboro un border object prima
dei
relativi core objects che lo contengono nel loro vicinato, al border obj è
assegnata
reachability distance pari ad infinito come fosse di rumore.
(3) Se il pattern, nella matrice di relazione, ha norma bassa, non considero il
suo vicinato ai fini dell'inserimento nella seed-list.
(4) Aggiorna la seed-list con i pattern del vicinato dell'ultimo punto; il
prossimo
pattern da elaborare sarà quello con minor reachability-distance presente in
seed-list;
se è vuota ne scelgo uno a caso.
(5) Ogni volta che trovo una steep-up la confronto con le steep-down precedenti;
si verifica se la coppia di aree di steep trovate rispetta la definizione di
cluster, in caso affermativo memorizzo gli indici di inizio e fine (rispetto
all'ordinamento
del vettore di reachability-distance) in un elemento dell'albero.
(6) Il costruttore si occupa di inizializzare le strutture dati e di avviare
la funzione che crea il grafico tipico di OPTICS
(7) Trova i cluster in base al contenuto del vettore delle reachability-
distances
il mib serve per limitare il numero di steep-down-areas da mantenere in memoria
(si veda l'articolo di presentazione di OPTICS)
(8) è il MIB globale cioè il massimo valore di reachability-distance tra la
fine dell'ultima steep area (upward o downward) e l'indice corrente
(9) Scorro tutte le csi downward areas (indice i)
(10) Il SoC dell'articolo (Figura 18b)
(11) L' EoC dell'articolo (Figura 18c)
```

```
(12) Vedi nell'articolo figura 18a
*/

////////////////////////////////////////////////////////////////////////////
////
// GeneralTree.h

//Davide Bolognesi 17-2-2006
//
class ListaSimple;

#pragma once
#include "globalfun.h"
#include "globalconst.h"
#include "resultOPTICS.h"
#include "listaSimple.h"

#include <iostream>
using namespace std;

extern long N;

struct Node{      //(1)
      long iniCl_ord;
      long fineCl_ord;
      Node * son;
      Node * bros;
      Node * father;
      short elab;
      int Num_ordine;
};

struct Clusters{
      long inizio;
      long fine;
};

class GeneralTree //(2)
{
      friend void scriviAppartenenza(const optics&,GeneralTree&,ostream&);
      friend ostream& operator<<(ostream& os,const GeneralTree& t);
private:
      bool linear;
      long N_ord;
      Clusters* cl;
      ListaSimple l;
      Node * root;
      resultOPTICS * res;
      void canc(Node*);
      void orderBestClust(Node* p);
      ostream& scrivi(ostream&,Node*,int)const;
      real_t goodness(Node*,long,long);
      void extractCrispList(Node*);
public:
      void insert(long begin,long end);
      GeneralTree(resultOPTICS*);
      ~GeneralTree(void);
      long* forceToN(int&,int);
};
/*
La classe implementa un albero generico cioè un albero in cui un nodo ha un
```

```
numero
qualunque di fratelli e di figli
(1) Un nodo dell'albero possiede i puntatori ai fratelli, ai figli, al padre.
Il numero elab è necessario alla funzione che estrae la lista degli N cluster
(2) Classe che implementa l'albero generico ed i metodi specifici per
l'elaborazione di
ROPTICS
*/

////////////////////////////////////////////////////////////////////////////
////
// GeneralTree.cpp

//Davide Bolognesi 17-2-2006
//
#include ".\generaltree.h"
#include "optics.h"

#include <iostream>
using namespace std;

extern const str_t numClustErr0;
extern const str_t prgNov;

GeneralTree::GeneralTree(resultOPTICS* ropt)
{
	res=ropt;
	root=NULL;
	N_ord=0;
	linear=false;
	cl=NULL;
}

GeneralTree::~GeneralTree(void)
{
	canc(root);
}

void GeneralTree::canc(Node* root){
	if(root==NULL)return;
	if(root->bros!=NULL){canc(root->bros);}
	if(root->son!=NULL){canc(root->son);}
	delete root;
}

void GeneralTree::insert(long begin,long end){  //(1)
	if(root==NULL){
		root=new Node;
		root->bros=NULL;
		root->father=NULL;
		root->son=NULL;
		root->iniCl_ord=begin;
		root->fineCl_ord=end;
		root->elab=0;
		root->Num_ordine=N_ord++;
	}
	else{
		Node* p=root;
		Node* q=NULL;
		Node* Nnew;
		do{
```

```
                  if(p->fineCl_ord<=end && p->iniCl_ord>=begin){ //(1a)
                        Nnew=new Node;
                        Nnew->son=p;
                        Nnew->father=p->father;
                        Nnew->iniCl_ord=begin;
                        Nnew->fineCl_ord=end;
                        Nnew->elab=0;
                        Nnew->Num_ordine=N_ord++;
                        Node* r=p;
                        while(r->fineCl_ord<=end&&r->iniCl_ord>=begin){
                              r->father=Nnew;
                              r=r->bros;
                              if(r==NULL){break;}
                        }
                        Nnew->bros=r;
                        if(q){
                              q->bros=Nnew;
                        }
                        else{
                              root=Nnew;
                              root->bros=NULL;
                        }
                        return;
                  }
                  q=p;
                  p=p->bros;
            }while(p);
            Nnew=new Node;
            Nnew->son=NULL;
            Nnew->bros=NULL;
            Nnew->Num_ordine=N_ord++;
            Nnew->iniCl_ord=begin;
            Nnew->fineCl_ord=end;
            Nnew->elab=0;
            q->bros=Nnew;
            Nnew->father=NULL;
      }
}

ostream& GeneralTree::scrivi(ostream& os,Node* rad,int level)const{
      while(rad){
            char* s;
            s=new char[level+1];
            s[level]='\0';
            for(int i=0;i<level;i++){
                  s[i]=' ';
            }
            if(rad->elab==1){
            os<<s<<"["<<rad->iniCl_ord<<","<<rad->fineCl_ord<<"]*"<<endl;
            }
            else{
                  if(rad->elab==-1){
                        os<<s<<"["<<rad->iniCl_ord<<","<<rad-
>fineCl_ord<<"]+"<<endl;
                  }
                  else{
                        os<<s<<"["<<rad->iniCl_ord<<","<<rad-
>fineCl_ord<<"]"<<endl;
                  }
            }
            if(rad->son!=NULL && linear==false){
```

```
            scrivi(os,rad->son,++level);
            level--;
            }
            rad=rad->bros;
      }
      return os;
}

ostream& operator<<(ostream& os,const GeneralTree& t){
      ostream& o=t.scrivi(os,t.root,0);
      return o;
}

long* GeneralTree::forceToN(int& n,int nc){     //(2)
      this->orderBestClust(root);                     //(2a)
      l.signN(nc);                                          //(2b)
      long* mem=new long[N];
      displayTime();
      cout<<prgNov<<endl;
      extractCrispList(root);                         //(2c)
      linear=true;
      Node* p=root;
      long j=0;
      root->iniCl_ord=0;
      while(p){
            if(p->bros==NULL){
                  p->fineCl_ord=N-1;
            }
            p->Num_ordine=j;
            for(long i=p->iniCl_ord;i<=p->fineCl_ord;i++){
                  mem[i]=j;
            }
            j++;
            p=p->bros;
      };
      return mem;
}

real_t GeneralTree::goodness(Node* p,long begin=0,long end=0){
      real_t sum=0;
      long num=1;
      if(p){
            begin=p->iniCl_ord;
            end=p->fineCl_ord;
            num=end-begin;
      }
            for(int i=begin;i<=end;i++){
                  sum+=res->reachability[i];
            }
      return (num^2)/sum;
}

void GeneralTree::extractCrispList(Node* p)     //(3)
{
      if(!p){
            return;
      }
      //cout<<*this;
      //cin.get();
      if(p->elab==1){
            Node* frat=p->bros;
```

```
if(frat && frat->elab==1){
      long bf,bp,ef,ep;
      bf=p->fineCl_ord+1;
      ef=frat->fineCl_ord;
      bp=p->iniCl_ord;
      ep=frat->iniCl_ord-1;
      real_t g1,g2;
      g1=goodness(NULL,bf,ef);
      g2=goodness(NULL,bp,ep);
      if(g2<g1){
            p->fineCl_ord=ep;
      }
      else{
            frat->iniCl_ord=bf;
      }
      extractCrispList(p->bros);
}
if(frat && frat->elab==0){
      if(frat->son){
            extractCrispList(frat->son);
            return;
      }
      else{
            if(frat->bros){
                  p->fineCl_ord=frat->iniCl_ord-1;
                  frat->bros->iniCl_ord=frat->fineCl_ord+1;
                  long bf,bp,ef,ep;
                  bf=frat->iniCl_ord;
                  ef=frat->bros->fineCl_ord;
                  bp=p->iniCl_ord;
                  ep=frat->fineCl_ord;
                  real_t g1,g2;
                  g1=goodness(NULL,bf,ef);
                  g2=goodness(NULL,bp,ep);
                  if(g2<g1){
                        p->fineCl_ord=ep;
                  }
                  else{
                        frat->bros->iniCl_ord=bf;
                  }
                  Node* x=frat;
                  p->bros=frat->bros;
                  delete x;
                  extractCrispList(p);
                  return;
            }
            else{
                  p->fineCl_ord=frat->fineCl_ord;
                  delete p->bros;
                  p->bros=NULL;
            }
      }
}
frat=p->bros;
if(!frat){
      if(p->father){
            Node* grandf=p->father->father;
            if(grandf){
                  Node* pad=p->father;
                  if(grandf->son==pad){
                        grandf->son=pad->son;
```

```
                        p->bros=pad->bros;
                        p->fineCl_ord=pad->fineCl_ord;
                        pad->son->iniCl_ord=pad->iniCl_ord;
                        Node * q=pad->son;
                        while(q){
                              q->father=grandf;
                              q=q->bros;
                        }
                        delete pad;
                  }
                  else{
                        Node* prec=grandf->son;
                        while(prec->bros!=pad){
                              prec=prec->bros;
                              if(!prec){
                                    cerr<<"Errore nella funzione
extractCList\n";
                                    exit(1);
                              }
                        }
                        prec->bros=pad->son;
                        p->bros=pad->bros;
                        p->fineCl_ord=pad->fineCl_ord;
                        pad->son->iniCl_ord=pad->iniCl_ord;
                        Node * q=pad->son;
                        while(q){
                              q->father=grandf;
                              q=q->bros;
                        }
                        delete pad;
                  }
                  extractCrispList(p);
                  return;
            }
            else{
                  if(p->father==root){
                        Node * tmp=root;
                        root=root->son;
                        p->bros=tmp->bros;
                        root->iniCl_ord=tmp->iniCl_ord;
                        p->fineCl_ord=tmp->fineCl_ord;
                        Node * q = root;
                        while(q){
                              q->father=NULL;
                              q=q->bros;
                        }
                        delete tmp;
                        extractCrispList(root);
                        return;
                  }
                  else{
                        Node * prec=root;
                        while(prec->bros!=p->father){
                              prec=prec->bros;
                        }
                        Node * pad=p->father;
                        prec->bros=pad->son;
                        p->bros=pad->bros;
                        p->fineCl_ord=pad->fineCl_ord;
                        pad->son->iniCl_ord=pad->iniCl_ord;
                        Node * q = root;
```

```
                                    while(q){
                                          q->father=NULL;
                                          q=q->bros;
                                    }
                                    delete pad;
                                    extractCrispList(root);
                              }
                        }
                  }
                  else{
                        return;
                  }
            }
      }
      if(p->elab==0){
            if(p->son){
                  extractCrispList(p->son);
                  return;
            }
            else{
                  if(p==p->father->son){
                        if(!p->bros){
                              if(p->father){
                                    p->father->son=NULL;
                                    Node * next = p->father;
                                    delete p;
                                    extractCrispList(next);
                              }
                              else{
                                    cerr<<"Errore inatteso\n";
                              }
                        }
                        else{
                              p->bros->iniCl_ord=p->iniCl_ord;
                              Node * next = p->bros;
                              p->father->son = next;
                              delete p;
                              extractCrispList(next);
                        }
                  }
                  else{
                        Node * q=p->father->son;
                        while(q->bros!=p){
                              q=q->bros;
                        }
                        if(!p->bros){
                              q->fineCl_ord=p->fineCl_ord;
                              delete q->bros;
                              q->bros=NULL;
                              extractCrispList(q);
                        }
                        else{
                              if(p->bros->son){
                                    extractCrispList(p->bros->son);
                                    return;
                              }
                              extractCrispList(p->bros);
                        }
                  }
            }
      }
```

```
}

void GeneralTree::orderBestClust(Node* p) //(4)
{
	if(!p){
		return;
	}
	if(p->son==NULL){
		l.inserisci(goodness(p),p);
		orderBestClust(p->bros);
	}
	else{
		orderBestClust(p->son);
		orderBestClust(p->bros);
	}
	return;
}

/*
(1) Inserisco un elemento. I vari cluster sono memorizzati come punto di inizio
e fine in
base all'ordinamento scelto da OPTICS; per come è fatta la funzione della classe
optics
che trova i cluster annidati, vengono trovati prima quelli interni poi quelli
esterni.
È sufficiente pertanto scorrere i fratelli della radice alla ricerca di uno o
più cluster
contenuti nel cluster che devo inserire (1a).
(2) Esegue le operazioni che portano all'estrazione dei cluster non annidati:
	(2a) Metto i nodi dell'albero in una lista ordinata in base alla goodness
	(2b) Segno i C nodi dell'albero (C = numero di cluster che voglio)
	che hanno maggiore goodness.
	(2c) Eseguo la extractCrispList e assegno una etichetta ai cluster.
(3) Funzione ricorsiva che, a partire dai cluster segnati (elab=1), modifica
l'indicazione dei punti di inizio e fine in modo da assegnare ogni punto ad uno
ed un solo cluster; data la complessità concettuale dell'albero generico
tale funzione esaurisce tutti i possibili casi (nodo con fratelli, figlio della
radice,
nodo senza fratelli da espandere fino a comprendere tutti i punti del padre
ed essere a lui sostituito...eccetera). Alla fine ho un albero con solo fratelli
della
radice, i cui indici di inizio e fine rappresentano i punti del cluster.
(4) Crea la lista ordinata delle foglie dell'albero; se anzichè le foglie
ordinassi
tutti i nodi il risultato è uguale.
*/

////////////////////////////////////////////////////////////////////////////
////
// MatrRel.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"

#include <iostream>
#include <fstream>
using namespace std;

class MatrRel
```

```
{
	friend ostream& operator<<(ostream& os,const MatrRel& m);
	real_t** rel;
public:
	MatrRel(void);
	~MatrRel(void);
	real_t norm1(int i)const;
	real_t ret_elem(int i, int j)const;
	real_t maxx();
	bool importFromASCII(ifstream& ing);
};

/* La classe memorizza la matrice di relazione; dato che è simmetrica ne
memorizzo
solo il triangolo sotto la diagonale. Se non è simmetrica la rendo tale.
*/

////////////////////////////////////////////////////////////////////////////
////
// MatrRel.cpp

//Davide Bolognesi 8-1-2006
//
#include ".\matrrel.h"
#include ".\globalfun.h"


#include <fstream>
using namespace std;

extern const str_t datAlloc;
extern void printpercent(int,int);
extern void printslash(int);
extern const str_t prgLoading;
extern long N;

MatrRel::MatrRel(void)  //(1)
{
	displayTime();
	cout<<"Allocazione struttura dati\n";
	rel = new real_t*[N-1];
	long tri=1;
	for(long i=0;i<N-1;i++){
		printpercent(i,N);
		rel[i]=new real_t[tri++];
	}
	cout<<"\r";
}

MatrRel::~MatrRel(void)
{
	for(long r=0; r<N-1; r++){
		delete rel[r];
	}
	delete[] rel;
}

real_t MatrRel::norm1(int i)const
{
	real_t sum=0;
		for(long j=0;j<N;j++){
```

```
                sum+=ret_elem(i,j);
            }
        return sum;
}

real_t MatrRel::ret_elem(int i, int j)const     //(2)
{
        if(i==j)return 0;
        if(i>j){
            return rel[i-1][j];
        }
        if(j>i){
            return rel[j-1][i];
        }
        cerr<<"Errore interno.\n";
        exit(-1);
}

ostream& operator<<(ostream& os,const MatrRel& t){
        cout<<"OutputStd"<<endl;
        for(int i=0;i<N-1;i++){
            for(int j=0;j<N-1;j++){
                os<<(float)t.ret_elem(i,j)<<" ";
            }
            cout<<endl;
        }
        return os;
}

bool MatrRel::importFromASCII(ifstream& ing)
{//(3)
        char text;
        real_t d;
        do{
            if(!ing){
                return false;
            }
            ing >> text;
        }
        while ( text != '[');
        displayTime();
        cout<<prgLoading<<endl;
        for(long i=0;i<N;i++){
            printpercent(i,N);
            for(long j=0;j<N;j++){
                if(ing >> d == 0 || ing.eof()){
                    return false;
                }
                if(j<i){//(4)
                    rel[i-1][j]=(d+rel[i-1][j])/2;
                }
                else{
                    if(i<j){
                    rel[j-1][i]=d;
                    }
                }
                ing.get(text);
                if(!ing){
                            return false;
                }
                while((text==' '||text=='\t')&& ing){
```

```
                        ing.get(text);
                        if(!ing){
                              return false;
                        }
                  }
                  if(text=='\n'){
                        ing.seekg(-1,ios::cur);
                  }
                  if(text!=';' && text !='[' && text!='/t' && text!=']'){
                        ing.seekg(-1,ios::cur);
                  }
            }
      }
      cout<<"\r";
      return true;
}

real_t MatrRel::maxx(){ //(5)
      if(rel==NULL || N==1){
            return -1;
      }
      else{
            real_t mass=0;
            for(long i=0;i<N-1;i++){
                  for(long j=0;j<i+1;j++){
                        if(rel[i][j]>mass){
                              mass=rel[i][j];
                        }
                  }
            }
            return mass;
      }
}
/*
(1) Riservo spazio per il triangolo inferiore della matrice
(2) Metodo di accesso agli elementi, accedo senza occuparmi del fatto che ho
memorizzato solo un triangolo della matrice
(3) Leggo la matrice da file in formato Matlab
(4) Memorizzo solo un triangolo; per gestire le non simmetrie faccio la media
degli elementi corrispondenti. Si dimostra che funziona meglio che prendere il
min
o il max o uno a caso.
(5) Calcolo il massimo tenendo conto che è simmetrica
*/

////////////////////////////////////////////////////////////////////////////
////
// resultOPTICS.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"

class resultOPTICS
{
public:
      long* originalIndex;
      real_t* reachability;
      resultOPTICS(void);
      ~resultOPTICS(void);
```

```
};

/*
Utilità di gestione del vettore di reachability-distances prodotto
da OPTICS
*/

////////////////////////////////////////////////////////////////////////////
////
// resultOPTICS.cpp

//Davide Bolognesi 8-1-2006
//
#include ".\resultOPTICS.h"
#include "globalconst.h"

extern real_t eps;
extern int minPts;
extern real_t csi;
extern long N;

#define INFINITO 2*(1/(1-csi))*eps

resultOPTICS::resultOPTICS()//(1)
{     originalIndex=new long[N+1];
      originalIndex[N]=-1;
      reachability=new real_t[N+1];
      reachability[N]=INFINITO;
}

resultOPTICS::~resultOPTICS(void)
{
      delete[] originalIndex;
      delete[] reachability;
}
/*
(1) Faccio spazio per un valore fittizio che servirà nel caso in cui
il vettore termina con una csi steep upward (altrimenti la funzione
di estrazione dei cluster non la vede)
*/

////////////////////////////////////////////////////////////////////////////
////
// ListaSimple.h

//Davide Bolognesi 8-1-2006
//
struct Node;

#pragma once
#include "globaltype.h"

#include <fstream>

using namespace std;

struct elem{
      elem* pun;
      Node* tree;
      real_t goodn;
};
```

```
class ListaSimple

{//(1)
private:
	elem* testa;

public:
	void writeElem();
	void inserisci(real_t,Node*);
	ListaSimple(void);
	~ListaSimple(void);
	void signN(int num);
};

/*
(1) Implementazione di una lista ordinata con l'aggiunta della funzione
che segna i primi N elementi.
Ogni elemento è un nodo di generalTree; in pratica metto ad 1 un flag
nei nodi con goodness maggiore
*/

////////////////////////////////////////////////////////////////////////////
////
// ListaSimple.cpp

//Davide Bolognesi 7-2-2006
//
#include ".\listasimple.h"
#include "globalfun.h"
extern bool logfile;
extern void writeErr();

extern void msgExit(char,int);
extern const str_t numClustErr0;

ListaSimple::ListaSimple(void)
{
	testa=NULL;
}

void ListaSimple::inserisci(real_t good,Node* nod){
	elem* p=new elem;//(1)
	p->goodn=good;
	p->tree=nod;
	if(!testa){
		testa=p;
		p->pun=NULL;
		return;
	}
	elem * q=testa;
	elem * r=NULL;
	while(q && q->goodn>good){
		r=q;
		q=q->pun;
	}
	if(!r){
		p->pun=testa;
		testa=p;
	}
	else{
```

```
            p->pun=r->pun;
            r->pun=p;
      }
}

ListaSimple::~ListaSimple(void)
{
      elem* q;
      while(testa!=NULL){
            q=testa->pun;
            delete testa;
            testa=q;
      }
}

void ListaSimple::writeElem(){
      if(testa==NULL){
            msgExit("Errore Strano2",1);
      }
      elem* q=testa;
      while(q!=NULL){
            cout<<q->goodn<<"["<<q->tree->iniCl_ord<<","<<q->tree-
>fineCl_ord<<"]"<<endl;
            q=q->pun;
      }
}

void ListaSimple::signN(int num)
{//(2)
      elem* p=testa;
      for(int n=0;n<num;n++){
            if(p){
                  p->tree->elab=1;
                  p=p->pun;
            }
            else{
                  if(logfile){
                        writeErr();
                  }
                  msgExit(numClustErr0,3);
            }
      }
}
/*
Gli elementi della lista sono i nodi dell'albero generico, in pratica memorizzo
una lista di puntatori ai vari elementi dell'albero
(1) Inserisco per goodness in ordine discendente
(2) Metto ad uno il flag elem nei nodi dell'albero
*/
////////////////////////////
// label.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"
#include <fstream>
using namespace std;

class Label
{     //(1)
```

```
	friend class optics;
private:
	int** corresp;
	int numReal;
public:
	bool readFromASCII(ifstream&);
	Label(void);
	~Label(void);
	void findRightLabel(ofstream&,int& numErr);
};

/*
(1) Classe che fornisce funzioni di utilità per il confronto delle etichette
trovate
dal clustering con quelle effettive, in particolare trova la giusta
corrispondenza
tra etichette
*/

////////////////////////////////////////////////////////////////////////////
////
// label.cpp

//Davide Bolognesi 17-1-2006
//
#include ".\label.h"
#include "globalconst.h"
#include "globalfun.h"
#include "globaltype.h"
#include <iostream>

extern long N;
extern const str_t usrNClustErr;
extern const str_t summary0;
extern const str_t summary1;
extern const str_t noMatrC;
extern bool confr=true;
extern int N_Clust;
extern bool lastClustNoise;
extern void printpercent(int,int);
extern void printslash(int);
extern void msgExit(const str_t usrErr, int nErr);

extern const str_t labLoadErr;

Label::Label(void)
{
	corresp=new int*[2];
	corresp[0]=new int[N];  //Reali
	corresp[1]=new int[N];  //ROPTICS
	numReal=0;
}

Label::~Label(void)
{
	delete corresp[0];
	delete corresp[1];
	delete corresp;
}

bool Label::readFromASCII(ifstream& ing)
```

```
{     //(1)
      char text;
      ing >> text;
      if(text!=';' && text !='[' && text!=' ' && text!='/t' && text!=']'){
                  ing.seekg(-1,ios::cur);
            }
      int b;
      do{
            if(!ing){
                  cout<<labLoadErr<<endl;
                  confr=false;
                  return true;
            }
            ing >> text;
      }
      while ( text != '[');
      for(int s=0; s<N; s++)
      {     ing >> b;
            if(b>numReal){
                  numReal=b;
            }
            corresp[0][s]=b;
            ing.get(text);
            while((text==' '||text=='\t')&& ing){
                        ing.get(text);
                  }
            if(text!=';' && text!=' ' && text!='/t' && text!=']'){
                  ing.seekg(-1,ios::cur);
            }
      }
      return true;
}

void Label::findRightLabel(ofstream& out,int& numErr)
{     //(2)
      if(confr){
                  int i;
                  long noise;
                  out << "CCprev=["<<endl;
                  for(i=0;i<N;i++){
                        out<<corresp[0][i]<<' ';
                  }
                  out<<';'<<endl;
                  for(i=0;i<N;i++){
                        if(corresp[1][i]<0){
                              cerr<<"Errore nella ricerca delle
corrispondenze!";
                              exit(0);
                        }
                        out<<corresp[1][i]<<" ";
                  }
                  out<<";\n];\n"<<endl;
                  int ** MC;
                  int * correspondance=new int[N_Clust];
                  MC = new int*[N_Clust];
                  for(int j=0;j<N_Clust;j++){
                        MC[j]=new int[N_Clust];
                        for(int y=0;y<N_Clust;y++){
                              MC[j][y]=0;
                        }
                  }
```

```
                for(int i=0;i<N;i++){
                        if((numReal>N_Clust-1 && !lastClustNoise)||
(numReal>N_Clust)){
                                msgExit(usrNClustErr,0);
                        }
                        noise=0;
                        if(corresp[0][i]>=N_Clust && lastClustNoise){
                                noise+=corresp[1][i];
                        }
                        else{
                                MC[corresp[0][i]][corresp[1][i]]++;
                        }
                }
                out<<"MatClass=[\n";
                int max=0;
                int errori=0;
                int curr=0;
                for(int i=0;i<N_Clust;i++){
                        max=0;
                        for(int j=0;j<N_Clust;j++){
                                out<<MC[i][j]<<" ";
                                if(MC[i][j]>max){
                                        errori+=max;
                                        max=MC[i][j];
                                        curr=j;
                                }
                                else{
                                        errori+=MC[i][j];
                                }
                                correspondance[i]=curr;
                        }
                        out<<';'<<endl;
                }
                out<<"];\n"<<endl;
                if(lastClustNoise){
                        out<<"Noise = "<<noise<<";\n"<<endl;
                }
                out<<"CorrOpt= [\n";
                for(int i=0;i<N_Clust;i++){
                        out<<' '<<correspondance[i];
                }
                out<<';'<<endl<<"];";
                double x=errori*100;
                x=x/N;
                cout<<endl;
                cout<<summary0<<errori<<summary1<<"("<<x<<"%)."<<endl;
                numErr=errori;
        }
        else{
                int i;
                out << "CCprev=["<<endl;
                for(i=0;i<N;i++){
                        out<<corresp[1][i]<<' ';
                }
                out<<endl<<"];"<<endl;
                cout<<noMatrC<<endl;
        }
}

/*
(1) Leggo le etichette giuste dal file di ingresso
```

```
(2) Confronto con quelle trovate da ROPTICS, trovo la giusta corrispondenza,
dice il numero e la percentuale di errore
*/

////////////////////////////////////////////////////////////////////////////
////
// orderSeeds.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"

class OrderSeeds
{
private:
      long n_elem;
      long dim;
      real_t* reachs;
      long*   indexs;
      long*   i_indexs;
public:
      void update (long index, real_t r);
      void getMin (long *index, real_t *r);
      void reorder(long);
      long n_seeds();
      OrderSeeds(long);
      ~OrderSeeds();
};
/*
Classe con funzioni di utilità che implementano il costrutto ordered seeds
*/
////////////////////////////////////////////////////////////////////////////
////
// orderSeeds.cpp

//Davide Bolognesi 8-1-2006
//
#include ".\orderseeds.h"
#define HANDLED -1
#define NOTHANDLED -2

OrderSeeds::OrderSeeds(long n)
{
  n_elem = 0;
  dim = n;
  n_elem = 0;
  reachs    = new real_t[dim];
  indexs    = new long[dim];
  i_indexs = new long[dim];
  for(long i=0; i<dim; ++i)
    indexs[i] = NOTHANDLED;
}

OrderSeeds::~OrderSeeds(void)
{
      delete[] reachs;
    delete[] indexs;
    delete[] i_indexs;
}
```

```
void OrderSeeds::reorder(long index) {
  long j;   //(1)
  real_t hf;
  long hi;
  if(index>0) {
    j = (index-1)/2;
    if(reachs[index]<reachs[j]) {
      while(index>0) {
      hf = reachs[index];
      reachs[index] = reachs[j];
      reachs[j] = hf;
      indexs[i_indexs[index]] = j;
      indexs[i_indexs[j]] = index;
      hi = i_indexs[index];
      i_indexs[index] = i_indexs[j];
      i_indexs[j] = hi;
      index = j;
      if(index>0) {
        j = (index-1)/2;
        if(reachs[index]>=reachs[j])
          index = 0;
      }
      }
      return;
    }
  }
  j = 2*index+1;
  while(j < n_elem) {
    if(j<n_elem-1)
      if(reachs[j+1]<reachs[j])
      ++j;
    if(reachs[j]<reachs[index]) {
      hf = reachs[index];
      reachs[index] = reachs[j];
      reachs[j] = hf;

      indexs[i_indexs[index]] = j;
      indexs[i_indexs[j]] = index;

      hi = i_indexs[index];
      i_indexs[index] = i_indexs[j];
      i_indexs[j] = hi;
      index = j;
      j = index*2+1;
    } else
      j = n_elem;
  }
}

void OrderSeeds::update(long index, real_t r) {
  if(indexs[index]!=HANDLED) {      //(2)
    if(indexs[index]==NOTHANDLED) {
      indexs[index] = n_elem;
      reachs[n_elem] = r;
      i_indexs[n_elem] = index;
      n_elem++;
      reorder(n_elem-1);
    } else {
      if(reachs[indexs[index]] > r) {
      reachs[indexs[index]] = r;
      reorder(indexs[index]);
```

```
      }
    }
  }
}

void OrderSeeds::getMin(long *index, real_t *r) {
//(3)
  *r = reachs[0];
  *index = i_indexs[0];
  n_elem--;
  indexs[i_indexs[0]] = HANDLED;
  if(n_elem>0) {
    reachs[0] = reachs[n_elem];
    i_indexs[0] = i_indexs[n_elem];
    indexs[i_indexs[0]] = 0;
    reorder(0);
  }
}

long OrderSeeds::n_seeds(){
      return n_elem;
}

/*
Gestisco la struttura oredered seeds che mi consente, in OPITCS, di prenedere
ogni volta
il punto più vicino. La struttura è memorizzata come uno heap
(1) Riordino lo heap dopo un inserimento
(2) Inserimento nella seed list
(3) Restituisce il pattern più vicino
*/

////////////////////////////////////////////////////////////////////////////
////
// csiSteepUp.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"
#include "resultOPTICS.h"

class csiSteepUp
{     //(1)
private:
      resultOPTICS * res;
      long dim;
      bool is_CsiSteepUp(long);
public:
      bool returnMin(long,long,long,long&);
      bool csiUpBegin(long,long&);
      csiSteepUp(resultOPTICS*,long);
      ~csiSteepUp(void);
};
/*
(1) Funzioni di utilità per la gestione delle csi upward areas
*/

////////////////////////////////////////////////////////////////////////////
////
// csiSteepUp.cpp
```

```
//Davide Bolognesi 8-1-2006
//
#include ".\csisteepup.h"

#define DEBUG
#ifdef DEBUG
#include <iostream>
using namespace std;
#endif

#define UNDEF -1

extern real_t eps;
extern int minPts;
extern real_t csi;

csiSteepUp::csiSteepUp(resultOPTICS* rif,long N)
{
	res=rif;
	dim=N;
}

csiSteepUp::~csiSteepUp(void)
{
}

bool csiSteepUp::csiUpBegin(long index,long& end){    //(1)
	if(index>dim){return false;}
	if(!is_CsiSteepUp(index)){return false;}
	long non_consecutive=0;
	long begin_non_steep=UNDEF;
	bool non_steep=false;
	int count=2;
	for(long i=index+1;i<dim;i++){
		if(res->reachability[i]<res->reachability[i-1]){
			if(begin_non_steep==UNDEF){
				begin_non_steep=i;
			}
		break;
		}
		if(!is_CsiSteepUp(i)){
			non_consecutive++;
			if(begin_non_steep==UNDEF){
				begin_non_steep=i;
			}
			else{
				if(non_consecutive>minPts){
					end=begin_non_steep;
					return true;
				}
			}
		}
		else{
			begin_non_steep=UNDEF;
			non_consecutive=0;
		}
	}
	if(begin_non_steep!=UNDEF){
		end=begin_non_steep;
	}
```

```
	else{
		end=dim-1;
	}
	return true;
}

bool csiSteepUp::is_CsiSteepUp(long index)
{	//(2)
	if(index>dim)return false;
	if(res->reachability[index]<=res->reachability[index+1]*(1-csi)){
		return true;
	}
	return false;
}

bool csiSteepUp::returnMin(long rechStart,long begin,long end,long& val){
	long last=-1;
	do{	//(3)
		if(res->reachability[begin]>res-
>reachability[rechStart]){last=rechStart;}
		else{break;}
		rechStart++;
	}while(begin<=end);
	if(last != -1){
		val=last;
		return true;
	}
	else{return false;}
}

/*
(1) Cerco il punto di fine della csi upward area (eseguirò la funzione per ogni
csi upward point)
(2) Valuta se è csi-steep upward poit
(3) Si utilizza questa funzione poiché i cluster non finiscono sempre nel punto
di fine di una csi steep upward. (Si veda nell'articolo: cerco il punto EoC,
Fig. 18c)
Per fare ciò confronto con i punti della csi downward corrispondente (da begin a
end).
*/

////////////////////////////////////////////////////////////////////////////
////
// csiSteepDown.h

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"
#include "resultOPTICS.h"

class csiSteepDown
{	//(1)
private:
	resultOPTICS * res;
	long dim;
	bool is_CsiSteepDwn(long);
public:
	bool returnMax(long,long,long,long&);
	bool csiDownBegin(long,long&);
	csiSteepDown(resultOPTICS*,long);
```

```
      ~csiSteepDown(void);
};


/*
(1) La classe fornisce funzioni di utilità per la gestione di
csi steep downward areas.
*/
////////////////////////////////////////////////////////////////////////////
////
// csiSteepDown.cpp

//Davide Bolognesi 8-1-2006
//
#include ".\csisteepdown.h"

#define UNDEF -1

extern real_t eps;
extern int minPts;
extern real_t csi;

csiSteepDown::csiSteepDown(resultOPTICS* rif,long N)
{
      res=rif;
      dim=N;
}

csiSteepDown::~csiSteepDown(void)
{
}

bool csiSteepDown::csiDownBegin(long index,long& end)
{           //(1)
      if(index>=dim-1){return false;}
      if(!is_CsiSteepDwn(index)){return false;}
      long non_consecutive=0;
      long begin_non_steep=-1;
      for(long i=index+1;i<dim;i++){
            if(res->reachability[i]>res->reachability[i-1]){
                  if(begin_non_steep==UNDEF){begin_non_steep=i;}
                  break;
            }
            if(!is_CsiSteepDwn(i)){
                  non_consecutive++;
                  if(begin_non_steep==UNDEF){begin_non_steep=i;
            }
                  else{
                        if(non_consecutive>minPts){
                                    end=begin_non_steep;
                                    return true;
                        }
                  }
            }
            else{
                  begin_non_steep=UNDEF;
                  non_consecutive=0;
            }
      }
      if(begin_non_steep!=UNDEF){
            end=begin_non_steep;
```

```
	}
	else{
		end=dim-1;
	}
	return true;
}

bool csiSteepDown::is_CsiSteepDwn(long index){
	if(index>=dim-1)return false;//(2)
	if(res->reachability[index]*(1-csi)>=res->reachability[index+1]){
		return true;
	}
	return false;
}

bool csiSteepDown::returnMax(long reachEnd,long begin,long end,long& val){
	long last=-1;	//(3)
	do{
		if(res->reachability[begin]>res-
>reachability[reachEnd]){last=begin;}
		else{break;}
		begin++;
	}while(begin<=end);
	if(last != -1){val=last; return true;}
	else{return false;}
}
/*
(1) Cerca il punto di fine della csi downward area in base alle defnizioni
teoriche
Ad es. non più di minPts punti consecutivi non steep downward.
(2) Dice se è uno csi-steep downward point
(3) Si utilizza questa funzione poiché i cluster non iniziano sempre nel punto
di inizio di una csi steep downward. (Si veda nell'articolo: cerco il punto SoC,
Fig. 18b)
Per fare ciò confronto con i punti della csi upward corrispondente (da begin a
end).
*/

////////////////////////////////////////////////////////////////////////////
////
// SetOfSteepDown.cpp

//Davide Bolognesi 8-1-2006
//
#pragma once
#include "globaltype.h"
#include "resultOPTICS.h"

extern real_t csi;

struct StDelem{
	StDelem* pun;
	long begin;
	long end;
	real_t mib;
	real_t Sd;
};

class SetOfSteepDown
{
	long elements;
```

```
      StDelem* head;
public:
      bool returnMib(long index,real_t& mib,long& begin,long& end);
      void update(resultOPTICS*,real_t);
      void add(long,long,real_t,real_t);
      SetOfSteepDown(void);
      void filter(real_t);
      ~SetOfSteepDown(void);
};

/*
Classe che gestisce la lista delle steep downward areas precedenti al punto in
cui ci si trova
*/

////////////////////////////////////////////////////////////////////////////
////
// SetOfSteepDown.cpp

//Davide Bolognesi 8-1-2006
//
#include ".\setofsteepdown.h"

#include <stdlib.h>
#include <iostream>
using namespace std;

SetOfSteepDown::SetOfSteepDown(void)
{
      head=NULL;
      elements=0;
}

bool SetOfSteepDown::returnMib(long index,real_t& mib,long& begin,long& end){
      if(head==NULL){   //(1)
            return false;
      }
      StDelem* p=head;
      for(long i=0;i<index;i++){
            p=p->pun;
            if(p==NULL)return false;
      }
      mib=p->mib;
      end=p->end;
      begin=p->begin;
      return true;
}

void SetOfSteepDown::filter(real_t mib){
      if(head==NULL)return;   //(2)
      StDelem* p=head;
      StDelem* next=p->pun;
      StDelem* last=NULL;
      do{
            if(((p->Sd)*(1-csi))<mib){
                  if(p==head){
                        delete head;
                        head=next;
                        if(head==NULL)return;
                        p=head;
                        last=NULL;
```

```
                        next=next->pun;
                }
                else{
                        delete p;
                        last->pun=next;
                        if(p==NULL)return;
                        next=next->pun;
                }
        }
        else{
                        last=p;
                        p=next;
                        if(p==NULL)return;
                        next=next->pun;
        }
    }while(p!=NULL);
}

void SetOfSteepDown::update(resultOPTICS* res,real_t current){
    if(head==NULL)return;
    StDelem* p=head;
    do{
        p->mib=max(p->mib,current);
        p=p->pun;
    }while(p!=NULL);
}

void SetOfSteepDown::add(long begin,long end, real_t reachStart,real_t mib){
    StDelem* q=new StDelem;
    q->begin=begin;
    q->end=end;
    q->mib=mib;
    q->Sd=reachStart;
    q->pun=head;
    elements++;
    head=q;
}

SetOfSteepDown::~SetOfSteepDown(void)
{
    StDelem* q;
    while(head!=NULL){
        q=head->pun;
        delete head;
        head=q;
    }
}
/*
(1) MIB: rappresenta il massimo valore di reachability-distance tra la fine
della
steep downward area in questione e l'indice corrente
Il MIB globale viene calcolato nella classe optics
(2) Filtro le aree che non interessano più: quelle il cui inizio moltiplicato
per (1-csi)
è minore del MIB globale (che tanto non avrebbero rispettato la sc1*
dell'articolo
paragrafo 4.3.2 in fondo).
*/
////////////////////////////////////////////////////////////////////////////
////
// globalconst.h
```

```
#ifndef GLOBALCONST_H
#define GLOBALCONST_H

#include "globaltype.h"
#endif

////////////////////////////////////////////////////////////////////////////
////
// globalconst.cpp

//Davide Bolognesi 8-2-2006
//
#include "globalconst.h"
//#define ENGL

// errori utente
#ifdef ENGL
//Inglese...
extern const str_t bar="=============================================";
extern const str_t prgTitle="ROPTICS v 1.51 derived from OPTICS";
extern const str_t prgLoading="Loading input data...";
extern const str_t usrErr0= "Errore generico";
extern const str_t datAlloc="Memory allocation";
extern const str_t done="...done!";
extern const str_t end="End.";
extern const str_t summary0="Clustering committed ";
extern const str_t summary1=" errors. ";
extern const str_t ursFileSynErrC="Syntax error in the C matrix of the input
file.";
extern const str_t summaryNoise0="There was at least ";
extern const str_t summaryNoise1=" points of noise";
extern const str_t prgNov="Extracting non overlap custers...";
extern const str_t noMatrC="End of clustering";
extern const str_t prgFR="Finding reachability-distances.";
extern const str_t prgFC="Finding nested clusters...";
extern const str_t prgUnnest="Aggregating clusters";
extern const str_t prgStrCl="The structure of the nested clusters is the
following...";
extern const str_t prgStrCl2="The structure of the non overlap clusters is the
following...";
extern const str_t prgFindEps="Finding eps";
extern const str_t prgPreOrd="Preliminary ordering";
extern const str_t usrErr1= "Error: Too few parameters.\n\nUsage: \noptics -m
[minPts] -n [matrix dimension] -c [csi] -g [num. cluster] [-v] [-h] [-r] <Matlab
file>\n\nWhere:\nMatlab file contain the relation matrix and the right
clustering";
extern const str_t usrErr2= "Error: No file name.";
extern const str_t usrUsage="Usage: \noptics -m [minPts] -n [matrix dimension]
-c [csi] -g [num. cluster] [-v] [-h] [-r] <Matlab file>\n\nWhere:\nMatlab file
contain the relation matrix and the right clustering";
extern const str_t logIni="\tLogfile generated by ROPTICS\n\n t-begin\t t-
end\tminPts\t   csi\tClassif. Rate";
extern const str_t usrFileErr="Error: Can't open file.";
extern const str_t usrMatrErr="Error: Not a simmetric matrix!!!";
extern const str_t usrDimErr="Error: Wrong dimension of input matrix!";
extern const str_t usrErrDim= "Error: You must give the dimension of the data.";
extern const str_t usrErrCsi= "Error: Csi must be less than one.";
extern const str_t usrErrNCl="Error: Number of cluster must be greater than 0.";
extern const str_t numClustErr0="Error: Can't find enough clusters: this dataset
doesn't contains so many clustrers ";
```

```
extern const str_t usrNClustErr="Error: The number of cluster of the original
dataset is greater than the number you specified.";
extern const str_t ursFileSynErr="Syntax error in the input file.";
extern const str_t usrErrNorm = "Error: Invalid max norm value.";
extern const str_t labLoadErr = "\tCan't load tha vector of the real partition;
comparison will not be performed";
#else
//Italiano...
extern const str_t bar="==============================================";
extern const str_t prgTitle="ROPTICS v 1.51 algoritmo derivato da OPTICS";
extern const str_t prgLoading="Lettura dati in ingresso...";
extern const str_t usrErr0= "Errore generico";
extern const str_t done="...fatto!";
extern const str_t end="Fine.";
extern const str_t summary0="ROPTICS ha commesso ";
extern const str_t datAlloc="Allocazione matrice relazionale";
extern const str_t summary1=" errori. ";
extern const str_t ursFileSynErrC="Errore di sintassi nella matrice di
ingresso.";
extern const str_t summaryNoise0="Ci sono almeno ";
extern const str_t summaryNoise1=" punti di rumore";
extern const str_t prgNov="Estrazione cluster non sovrapposti...";
extern const str_t noMatrC="Fine del clustering";
extern const str_t prgFR="Ricerca delle reachability-distance.";
extern const str_t prgFC="Ricerca cluster annidati...";
extern const str_t prgUnnest="Aggregazione cluster";
extern const str_t prgStrCl="La struttura dei cluster annidati e' la
seguente...";
extern const str_t prgStrCl2="La struttura dei cluster non sovrapposti e'...";
extern const str_t prgFindEps="Ricerca eps";
extern const str_t prgPreOrd="Ordinamento preliminare";
extern const str_t usrErr1= "Errore: Parametri insufficienti.\n\noptics  -n
dimensione-matrice -g num-cluster [-m minPts] [-c csi]  [-1 min-norma] [-v] [-h]
[-r] <m-file>\n\nIn cui:\nm-file contiene la matrice di relazione ed il vettore
con le appartenenze.";
extern const str_t usrErr2= "Errore: Occorre specificare il file di input.";
extern const str_t usrUsage="Uso: \noptics  -n dimensione-matrice -g num-cluster
[-m minPts] [-c csi]  [-1 min-norma] [-v] [-h] [-r] <m-file>\n\nIn cui:\nm-file
contiene la matrice di relazione ed il vettore con le appartenenze.";
extern const str_t logIni="\tLogfile generato da ROPTICS\n\n t-inizio\t t-
fine\tminPts\t   csi\ttasso classificazione.";
extern const str_t usrFileErr="Errore: Impossibile aprire il file.";
extern const str_t usrMatrErr="Errore: Matrice non simmetrica!!!";
extern const str_t usrDimErr="Errore: Dimensioni sbagliate della matrice di
ingresso!";
extern const str_t usrErrDim= "Errore: Occorre specificare la dimensione dei
dati in ingresso.";
extern const str_t usrErrCsi= "Errore: Csi deve essere minore di uno.";
extern const str_t usrErrNCl="Errore: Il numero dei cluster deve essere maggiore
di zero.";
extern const str_t numClustErr0="Errore: Impossibile trovare tutti i cluster; il
dataset non ne contiene abbastanza (o non e' possibile distinguerli) ";
extern const str_t usrNClustErr="Errore: Il numero di cluster del data set
originale è maggiore del numero specificato.";
extern const str_t ursFileSynErr="Errore di sintassi nel file di ingresso.";
extern const str_t usrErrNorm = "Errore: Valore massimo della norma non
valido.";
extern const str_t labLoadErr = "\tImpossibile caricare il vettore delle
appartenenze, il confronto non sara' effettuato";
#endif
```

```
/*
Etichette usate per l'output delle applicazione
*/

////////////////////////////////////////////////////////////////////////////
////
// globalfun.h

//Davide Bolognesi 8-1-2006
//
class optics;
class generalTree;

#pragma once
#include "globaltype.h"
#include "optics.h"
#include "generalTree.h"
#include <time.h>

#include <iostream>
using namespace std;

extern void msgExit(char *,int n);
extern real_t maximum(real_t**,long);
extern void scriviAppartenenza(const optics&,GeneralTree&,ostream&);
extern void displayTime(void);
extern inline void beep(int n);
extern void printslash(int);
////////////////////////////////////////////////////////////////////////////
////
// globalfun.cpp

//Davide Bolognesi 8-1-2006
//
#include "globalfun.h"

void msgExit(char* msg,int n){
      cerr<<msg;
      exit(n);
}

void displayTime()
{   char ora[9];
    _strtime( ora );
      cout << "<" << ora << "> ";
}

inline void beep(int n)
{     for(int i=0; i<n; i++) cout << BEEP_CHAR; }

size_t randint( size_t min, size_t max)
{     static size_t res = 1;
      size_t r =  size_t ( floor( ( real_t(rand())/(RAND_MAX+1E-4) )*(max+1-
min))+ min );
      res = (res + r) % (max+1-min) + min;
      return res;
}

void printpercent(int i,int N){
      cout.precision(0);
      cout<<"\r\t\r"<<(float)i/N*100<<"%   ";
```

```
}
void printslash(int code)
{
      if (code!=0){
            cout << "\r " << SLASH[randint(0,3)] << "   ";
            cout.flush();
      }
      else{
        cout << "\r";
      }
}

/*
Funzioni di utilità usate nell'applicazione
*/

////////////////////////////////////////////////////////////////////////////////
////
// globaltype.h

#ifndef GLOBALTYPE_H
#define GLOBALTYPE_H

#include <float.h>
#include <valarray>
#include <limits.h>


      typedef float                              real_t;
                        //(1)---------------- float
      typedef char*                              str_t;
                        //(3)
//    typedef unsigned int               size_t;
                  //(2)

      //const real_t INFINITY = LDBL_MAX;
            //     +++++++++++++++++ double
      //const real_t INFINITESIMAL = LDBL_MIN;
            //     +++++++++++++++++ double
      const real_t INFINITY = FLT_MAX;
            //     ---------------- float
//    const real_t INFINITESIMAL = FLT_MIN;
            //     ---------------- float
      const char   BEEP_CHAR = 7;
      const char   SLASH[4]={'|','/','-','\\'};
      //(6)
#endif
```

Bibliografia

[1] Ester-Kriegel-Sander-Xu, DBSCAN, *A Density-Based Algorithm for Discovering Clusters*, KDD-96.

[2] Ankerst-Breunig-Kriegel-Sander, *OPTICS: Ordering Points To Identify the Clustering Structure*, SIGMOD-99.

[3] A.K. Jain, M.N. Murty, P.J. Flynn, *Data Clustering: A rewiew, ACM Computing Surveys*, Vol. 31 No. 3, Settembre 1999

[4] J.C. Bezdek, J. Keller, R. Krisnapuram, N.R. Pal, *Fuzzy Model and Algorithms for Pattern Recognition and Image Processing,*
Kluwer Academic Publishing, Boston, 1999

[5] P. Corsini, B. Lazzerini, F. Marcelloni, "*A new fuzzy relational clustering algorithm based on the fuzzy C-means algorithm,*" Soft Computing, Vol. 9, No. 6, 2005, pp. 439-447.

[6] M. Roubens, "*Pattern classification problems and fuzzy sets*", Fuzzy Sets and Systems, Vol. 1, 1978, pp. 239-253.

[7] M.P. Windham. "*Numerical classification of proximity data with assignment measures."* Journal of Classification, Vol. 2, 1985, pp. 157-172.

[8] R.J. Hathaway, J.W. Davenport, J.C. Bezdek. *"Relational duals of the c-means clustering algorithms"* Pattern Recognition, Vol. 22, 1989, pp. 205-212.

[9] R.J. Hathaway and J.C. Bezdek, "*NERF c-means: Non-Euclidean relational fuzzy clustering*" Pattern Recognition, Vol. 27, 1994, pp. 429-437.

[10] L. Kaufman, P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York: Wiley, 1990.

[11] Krishnapuram R, Joshi A, Nasraoui O, Yi L, *"Low-complexity fuzzy relational clustering algorithms for web mining"* IEEE Transactions on Fuzzy Systems, Vol. 9, No. 4, 2001, pp. 595-607.

[12] R.N. Davé, S. Sen, *Robust Fuzzy Clustering of Relational Data, IEEE Transactions on Fuzzy Systems*, Vol. 10, No. 6, 2002, pp. 713-727.

[13] Bjarne Stroustrup *C++, linguaggio, libreria standard, principi di programmazione*, Addison-Wesley – Milano 2000.

[14] S. Bennet, J. Skelton, K. Lunn, *Introduzione a UML,* McGraw-Hill 2002

[15] B.Lazzerini, *Algoritmo C-means,* Appunti per il corso di Sistemi Intelligenti 2004

[16] Haojun Sun, Shengrui Wang, Qingshan Jiang, *A new validation index for determining the number of clusters in a data set*

[17] Kaufman, L. and P.J. Rousseeuw, *Finding Groups in Data,* John Wiley & Sons, New York, 1990.

[18] S. Ricciarelli, *Algoritmi di data clustering basati sulla densità,* Università di Bologna 2001

[19] M. Steinbach, L. Ertöz, V. Kumar, *The Challenges of Clustering High Dimensional Data*

[20] K. Beyer, J. Goldstein, R. Ramakrishnan, U.Shaft, *When is 'Nearest Neighbor' meaningful?*

[21] M. G. Cimino, B. Lazzerini, F.Marcelloni, *A Novel Approach to Fuzzy Clustering based on a Dissimilarity Relation extracted from Data using a TS System*